PARALLEL IMPLEMENTATIONS OF SELECTED IMAGE PROCESSING TECHNIQUES

by

Christopher Turner

TR93-077 May 1993

This is an unaltered version of the author's
M.Sc.(CS) Thesis

Faculty of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, N.B.
Canada E3B 5A3

Phone: (506) 453-4566
Fax: (506) 453-3566

Abstract

In this thesis we present parallel implementations of one local (convolution) and one global (regular moment extraction) image processing technique on a multi-transputer system. Issues relevant to implementation design, including computational algorithm selection, initial data pass, and topology selection are discussed. Linear speedups in the convolution implementations are observed whereas the efficiency of the regular moment programs decreases as the number of transputers increases. Analysis of the implementations including parallel time complexity functions and observations about data passing and topology selection is given. Two theoretical performance models based on the implementations closely match empirical timing results.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisors Professor V. C. Bhavsar and Professor P. Pochec. Partial financial support through the NSERC operating grants OGP00089 and OGP0105827 of Professor Bhavsar and Professor Pochec is also acknowledged. The U.N.B. supercomputing group deserves special thanks, especially Brian d'Auriol, the 'transputer pioneer' at U.N.B. Mr. d'Auriol's groundwork in the area during 1991 and willingness to answer my questions greatly hastened completion of this thesis.

I am grateful to U.N.B. students Eric Drummie and Derrick Weaver for putting me up during the early winter of 1991. I would also like to thank Fredericton soccer players Geoff Downey, Tom Hanley and Jean LeBlanc for their support and encouragement. Most importantly of all, I would like to thank my family, on whose strength I have drawn every day while in New Brunswick.

Chapter 1

Introduction

One of the great scientific challenges of the 1990's seems to be keeping pace with man's ever increasing appetite for computational power. It seems that as soon as new, faster computers become available, new applications are found that make them seem inadequate. In addition, in recent years it has become increasingly obvious that the era when orders of magnitude improvement in processor speed can be expected is coming to an end. The laws of physics dictate that there is a limit to how powerful traditional single processor computers can be. The only solution to the problem seems to be to use computers with more than one processor. This is known as parallel processing. While many advances have already been achieved in this area, it is clear that man has only begun to discover the potential of parallel computing.

The T800 transputer is a VLSI processor marketed by INMOS Ltd. Transputers can be used individually or linked together into reconfigurable networks. Applications can be programmed on these networks using special transputer programming languages such as Parallel C. These networks form relatively inexpensive and flexible parallel computers. Transputers are often mounted in boards which are connected to host computers. The B008 motherboard, also released by INMOS, can house up to ten T800 transputers.

One area of computer science that can benefit from the computational power

of parallel processing is image processing. Image processing deals with the transformation and analysis of pictures. Real-time image processing applications require operations to be executed on the thousands of pixels that make up image frames many times each second. One operation commonly performed in such applications is feature extraction. Feature extraction is the act of finding distinguishing features or measures in images. Features can be either local or global. Convolution, which involves the passing of a mask over an image is a local feature extraction process. Statistical regular moments, on the other hand, are global image features.

1.1 Thesis Objective

The feasibility of transputer based image processing systems has been discussed in [13, 19]. The objective of the thesis is to develop efficient parallel implementations of the image processing techniques of convolution and regular moments on a B008 motherboard containing nine T800 transputers programmed with Parallel C. Our main criterion for judging the efficiency of an implementation will be its execution time. Its memory requirements will also be considered. Special attention will be given to implementation performance as image size and number of processors increases.

We acknowledge that just as moments and convolution are only examples of image processing techniques, using the B008 and Parallel C is only one method of computing with transputers. Therefore, we would like to abstract from the special considerations of our chosen image processing techniques and transputer model and give some attention in our research to the more general issue of efficient image processing on transputers. Specifically, we would like to discover some principles and ideas that are applicable to any transputer based image processing application. Ideally, our research might lay the foundation for a *generic* efficient image processing system on message passing, distributed memory multiprocessor systems.

An alternative approach to the research is to use the reconfigurability of transputer networks to find the best implementations for regular moments and convolution on certain topologies commonly found in parallel computing (*e.g.*, ring, mesh, binary tree). In this approach, transputers become an experimental model for image processing on parallel computers. We believe, however, that transputers are appropriate themselves for image processing and that the issues related to their use for image processing applications are sufficiently interesting and challenging to merit investigating image processing on transputers for its own sake.

1.2 Strategy

Multi-transputer programming is sufficiently complicated to challenge even the best programmer. A starting point is necessary from which parallel programs can be developed. Our approach is to begin by implementing more traditional, single process programs on a single transputer in Parallel C. Next we will program multi-process programs and run them on a single transputer. These programs will be finally converted to run on multiple transputers. When designing the multi-transputer programs, we intend to take into consideration the following issues: 1) topology selection; 2) data passing techniques; and 3) parallel computational algorithm selection.

1.3 Thesis Structure

The second chapter gives the reader a broad introduction to parallel processing. Emphasis is given to transputers and Parallel C. Chapter 3 introduces image processing, focusing on our transputer based image processing system and the techniques of regular moments and convolution. Chapter 4 gives timing results for our single transputer implementations. Chapter 5, the longest in the thesis, gives timing results for and analyzes our eight multi-transputer programs. Performance modelling for two of those

implementations is presented in Chapter 6. The thesis ends with a conclusion which summarizes our work and discusses some possible avenues for future research.

Chapter 2

Parallel Processing

2.1 Introduction

The computational power of any traditional, single processor computer is largely determined by three factors: the speed of its CPU, the speed and extent of its main memory and the speed and size of the channel between its CPU and main memory. The last factor, the classic 'Von Neuman bottleneck' is the most restrictive to today's single processor computers [6]. Parallel computers, computers which incorporate parallel processing, circumvent at least one of the above restrictions. In this chapter we discuss parallel processing. It should be noted that parallel processing is an extremely diverse, complicated branch of computer science. A comprehensive examination of it would likely fill several volumes. Our approach in this chapter is to introduce the reader to its most important aspects, paying special attention to the ones needed to understand the research we later present. For more information about a given topic, the reader should consult the listed reference.

We begin the chapter with an examination of the various types of parallel architectures. Next, parallelization techniques of sequential algorithms are discussed. Some useful terms and definitions related to parallel processing are discussed in the following section. A section devoted to transputers follows these definitions. Parallel

C, the language we have used to program the transputers, is next described. Some concluding remarks close the chapter.

2.2 Parallel Architectures

There is a bewildering number of different kinds of computers in the 1990's. Great differences exist in architecture and technology from one machine to another. While advances in computer technology seem to occur virtually every day, (consider, for example, recent breakthroughs in chip design and makeup) the architectures themselves have been relatively unchanged. In this section we attempt to describe the various architectures of current computers and give examples of machines that use them. There are four main factors to consider when classifying the architecture of a parallel computer. The first is how many data streams it has. The second is whether or not the computer's processors can execute different instructions at one time. The computer's memory type is also relevant. Lastly, how a parallel computer's processors communicate with each other must be considered when classifying it. We next describe each issue in more detail.

2.2.1 Data Streams

The processors of a parallel machine can either access data through the same path or can have their own path to data. In the first case, the parallel machine is said to be a Single Data Stream machine. Those machines whose processors have their own path to data are called Multiple Data Stream computers.

2.2.2 Instruction Streams

If a computer can execute only one instruction at once, it is referred to as a Single Instruction Stream machine. Obviously, all single processor sequential computers

are single instruction stream machines. Some multiple processor machines contain many simple processors which all perform the same action at the same time (Single Instruction Stream Multiple Data Stream machines). Others have more sophisticated processors which can do different operations simultaneously. The latter are called Multiple Instruction Stream computers.

2.2.3 Memory Type

Some parallel computers have a single memory which is accessed by all of its processors. These machines are said to be shared memory machines. On the other hand, the processors of some machines have their own memory space. They are called distributed memory machines. It is possible for both types of memory to be found in one parallel computer.

2.2.4 Communication Method

If the processors of a parallel computer communicate and share data by means of a central shared memory, the computer is referred to as a *tightly coupled* machine. If, on the other hand, its processors communicate directly with each other through communication links, without a shared memory, the parallel machine is *loosely coupled.* In the former case, the processors must be carefully controlled so that memory contention is resolved and no processor can access data that another is altering. In the latter case, the computer's processors can run independently and asynchronously, but must have message passing capability for communication [25].

The above four factors can be combined to get many kinds of parallel machines. Conventional single processor, single memory computers are classified as Single Instruction, Single Data (SISD) machines. Most arrays of VLSI processors are Single Instruction, Multiple Data (SIMD) distributed memory computers. Vector supercomputers are also usually SIMD machines. Transputers, as we shall see, are loosely

coupled Multiple Instruction Stream Multiple Data Stream (MIMD) computers.

2.3 Parallelization Techniques

A parallelization technique is a method of doing a problem on a parallel computer. The three major ways of doing problems on parallel machines are data parallelism, instructional parallelism and processor farming. In this section, we discuss each.

2.3.1 Data Parallelism

The conceptually simplest parallelization technique is data parallelism. In the data parallelism method, each of many processors executes the same instructions on part of a large amount of data. This is the technique used by vector supercomputers.

2.3.2 Instructional Parallelism

In contrast with data parallelism, in instructional parallel programs the processors work together to perform a computation on the same data. The processors usually execute different operations on the data. The best example of instructional parallelism is pipeline processing. In this case, each processor in a line of processors performs a separate action on the data and then passes it to the next processor in the line.

2.3.3 Processor Farming

Another common approach is processor farming. In farming, a central supervisory 'farmer' processor dynamically allocates tasks for several 'worker' processors. The workers perform the tasks they are allocated and send the results back to the farmer. The process continues until all necessary work is done.

2.4 Terms and Definitions

In this section, we present some of the terms and definitions associated with parallel processing.

2.4.1 Speedup

The speedup of a parallel implementation with P processors is defined to be the time taken for the task on one processor divided by the time taken on the parallel implementation. The concept of speedup is formalized in Equation 2.1.

$$S(P) = T(1)/T(P) \tag{2.1}$$

A linear speedup for an implementation is preferred since it usually indicates that the implementation has overhead directly in proportion to the number of processors. However, in many practical implementations, the overhead usually increases more than linearly as the number of processors increases, resulting in speedup curves which flatten out as the number of processors increases.

2.4.2 Efficiency

The efficiency of a P processor parallel implementation is defined as the speedup of the implementation divided by P. Equation 2.2 expresses efficiency in formal terms.

$$E(P) = S(P)/P \tag{2.2}$$

The efficiency of an implementation can never exceed one. Implementations with efficiencies close to one are preferable. However, an inefficient implementation of one parallel algorithm may still be faster than an efficient implementation of a slower parallel algorithm. It is a measure of how well the implementation uses the hardware, and not how good the algorithm used by the implementation is.

2.4.3 Parallel Processor Cost

For a P processor parallel implementation, this is defined as the product of the time taken for the task and P, the number of processors. This is stated symbolically in Equation 2.3.

$$C(P) = T(P) * P \tag{2.3}$$

This measures the unit cost of the parallel implementation.

2.4.4 Granularity

There are three types of granularity, *problem granularity*, *task granularity* and *computer granularity*.

Problem Granularity

The granularity of a problem refers to the size of the parts into which it may be subdivided for parallel processing. A problem is said to be *fine grained* if these parts are relatively small, *large grained* if they are large.

Task Granularity

The granularity of a task or process measures the amount of computation that can be performed without required communication/synchronization with other tasks.

Computer Granularity

Computer granularity measures the ratio of the number of processors on a parallel machine to their power. A *large grained* parallel machine has a few very powerful processors. Most supercomputers are large grained. On the other hand, a *fine grained* computer has numerous slower, simple processors. A VLSI chip with many processors is a good example of a fine grained parallel machine. Best results usually are achieved

when the granularity of the desired problem is close to the granularity of the machine upon which it is to be executed.

2.5 Transputers

In 1984 INMOS Ltd. of Great Britain began development of a VLSI processor based on a novel idea. Until then, most programming languages had been designed to run on a given computer architecture (usually SISD). INMOS's idea was to design its processor to implement an existing parallel programming language. The processor is called a transputer (an acronym for TRANSistorized comPUTER). The language is called OCCAM. OCCAM and its successor OCCAM 2 implement the Communicating Sequential Process model of parallel computation.

2.5.1 Communicating Sequential Processes

Communicating Sequential Processes (CSP's), were proposed in 1978 by Hoare [11]. A CSP is one of at least two simultaneously executing processes which communicate and synchronize themselves by means of message passing through channels. The processes successively perform an operation on data and pass the altered data on to the next process through the channel connecting them.

2.5.2 Transputer Networks

It is possible to use transputers by themselves, but they are computationally most effective when many are connected together into a network. In the latter case, the transputers are usually placed in a board such as the IMS B008 (see Section 2.5.7) that is connected to a host computer. A special transputer called the root is connected to the host and can access its monitor, disk drive and keyboard. Transputer networks

form loosely coupled parallel machines implementing the MIMD distributed memory parallel architecture. This makes them ideal for implementing both data parallel and instructional parallel algorithms. Since most such networks contain relatively few transputers (usually less than 128), and since each transputer is quite powerful computationally, they are considered to be relatively large grained machines. The networks are created by connecting pairs of the high speed bidirection links on each transputer together. These links are used for data communication and synchronization between the two connected transputers. Many different network topologies can be formed in this way. Once created, these networks can be programmed using OCCAM2 or another transputer programming language such as Parallel C. Special operating systems (*e.g.* HELIOS) are available that are designed for transputer environments [21].

2.5.3 The IMS T800 Transputer

Released in October 1988, the T800 is a powerful transputer currently available from INMOS. It has up to 4 Megabytes of RAM and a clock rate of 30MHz. The T800 has a 32 bit CPU and a 64 bit floating point unit. Its four bidirectional links send data at a rate of 2.35 Mbytes per second. Its peak performance is rated at 2.25 MFLOPS for a 30MHz clock rate. Graphics support is available.

2.5.4 The IMS C012 Communication Device

The C012 performs parallel to serial and serial to parallel conversions back and forth from the main host bus to the root transputer on the B008. The IMS S708B device driver is a program on the host which assists in this data transfer.

2.5.5 The IMS C004 Link Switch

The C004 is a crossbar switch used to reconfigure the B008 motherboard. It has 32 link ports which can be connected together for this purpose.

2.5.6 The IMS T222 Network Configuration Transputer

This 16 bit transputer is responsible for resetting the C004 crossbar switch in order to change the topology of the B008.

2.5.7 The IMS B008 Motherboard

The B008 has ten slots for TRAMS or TRAnsputer ModuleS. The first slot contains the root transputer. Each slot has four link connections which are joined to the links of the transputer placed in the slot. The ten slots are permanently connected into a pipeline using two of the links of each slot. The other two links are connected to a C004 crossbar switch. The notable exception is the first slot. Its first link is connected to a C012 communication device. The second link is connected to a T222 network configurer. Its third link is connected to the second slot. The fourth and final link is connected to the C004. Figure 2.1 shows the B008 configuration.

During network reconfiguration, the C004 gets the information about how to connect its links from a network configuration program which executes on the T222. The program receives two host files containing Module Motherboard Software (MMS) code. The first file describes the B008 hardware and its hardwire (permanent) connections. The second lists the softwire (temporary) link connection information needed for the new topology. Using this information, the program sends the C004 the low level linking commands necessary to complete the topology and reset the B008. Network configuration is done by executing the *tnconfig* command on the two MMS files. This causes the *iserver* to send the information in the files to the T222.

2.5.8 The IMS T9000 Transputer

The eagerly awaited T9000 transputer is the centrepiece of the next generation of transputer technology currently in development at INMOS. It features an order of magnitude improvement in link and processor speeds. Its most interesting new aspect

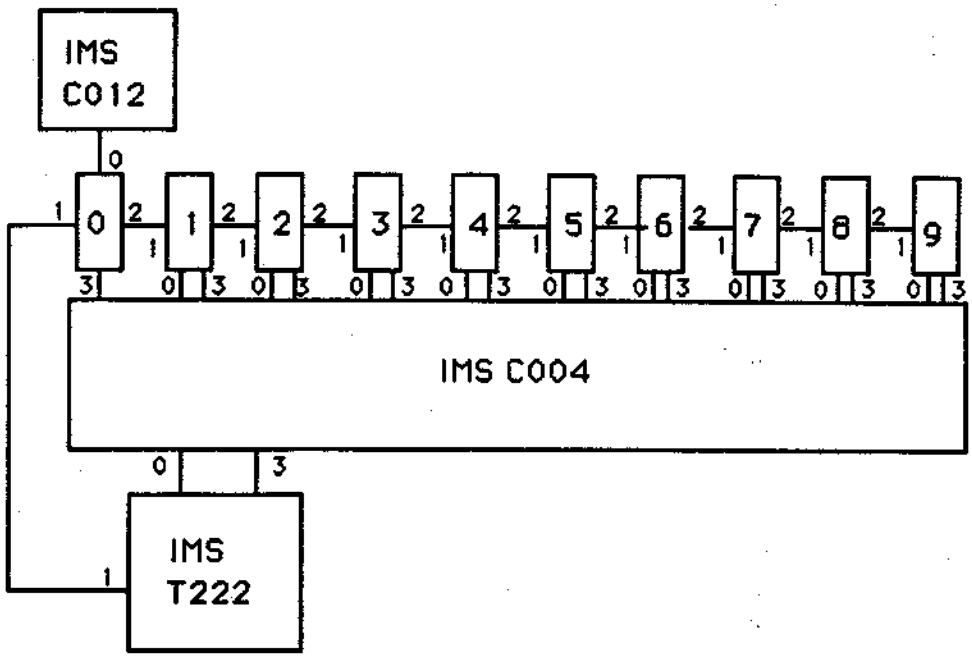

Figure 2.1: The B008 Motherboard

though, is its virtual channel processing capability. This will automate the routing of data packets. Previously the programmer had to perform this in software. Latest estimates indicate that the technology will be available in the summer of 1993.

2.5.9 Optimal Transputer Programming Model

In order to exploit the full potential of the multi-transputer parallel model, the following method can be used to program each transputer. One asynchronous routing process is dedicated to control each unidirectional transputer link. In addition, one or more asynchronous processes are created to perform all computation on the transputer. The routing processes control data flow to and from the transputer and to and from the computational processes. The computational processes are analogous to procedures in a conventional single processor program. They act as Communicating Sequential Processes. Each one performs a separate computational task, then passes

the resulting data to the next computational process or to a routing process and then terminates. Data communication and synchronization among all the processes is achieved by means of soft channels. These unidirectional channels between processes are analogous to the hard transputer link channels which connect transputers. Just as any two transputers which must communicate must be connected by a link, so too must any two processes which communicate be connected by a soft channel. Routing processes may be connected to each other or to computational processes. Obviously, the computational processes must also be connected by a channel. Figure 2.2 gives a schematic diagram for this method of programming a transputer. The use of this programming method not only enables parallel transputer implementations to run as fast as possible but also makes it easier to avoid deadlock in data passing operations between transputers.

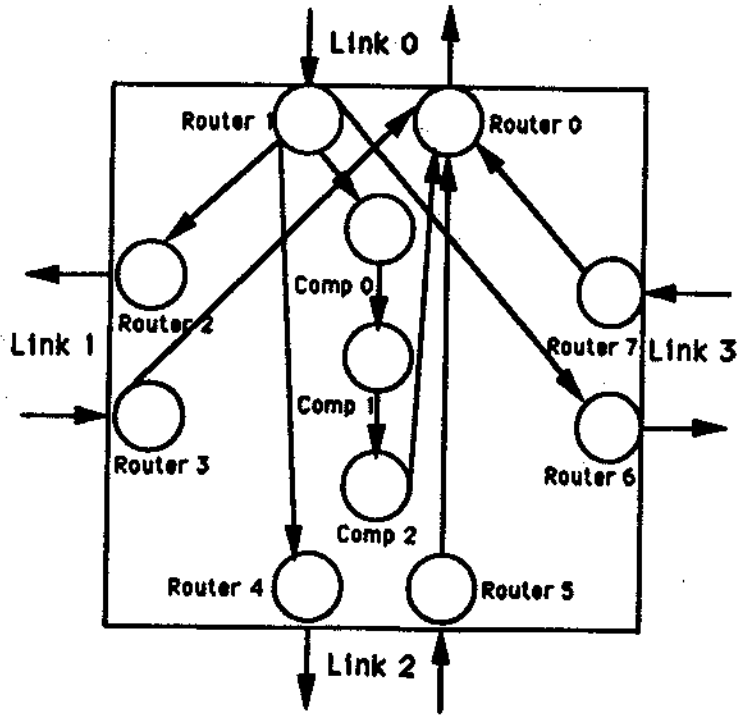

Figure 2.2: Optimal Transputer Programming Model

2.6 Parallel ANSI C

Just as OCCAM 2 evolved from OCCAM, Parallel ANSI C evolved from an earlier language developed by 3L Ltd. It is a superset of ANSI standard C, which in turn standardized and improved upon Kernighan and Ritchie's C. Its additional features allow for parallel processing on transputers using the CSP model. Parallel ANSI C programs consist of one or more source files and an optional configuration source file. These are described below.

2.6.1 Source Files

Parallel ANSI C source files are quite similar to ANSI C source files. They have the *c* file extension usual for C programs. In addition, each must include a *main()* function returning an *int*. The similarities end there, however. During compilation, each source file is either linked with the full or the reduced library of functions to produce a linked unit process binary. (See Section 2.6.3 for a description of the compilation procedure.) There must be one linked unit connected to the host in each Parallel C program. Linked units connected to the host must be linked with the full library. They may access the command line parameters *argv* and *argc*. Such units may also use the entire *io* library including such functions as *printf* and *fgetc*. Other linked units cannot use these functions.

Within each source file, local synchronous and asynchronous processes can be created and run at high or low priority. Parallel C allows for the creation of local 'soft' channels for communication and synchronization between these local processes as in the CSP model. The *get_param()* function can be used to map a soft channel onto a connection between two linked units defined in the *interface* statement in the configuration source file of the implementation as described in Section 2.6.2. In the case of programs consisting of more than one linked unit (or, equivalently, more than one source file) the units can be connected for data communication. The units may

also be configured to run on a network of transputers. The last two operations are accomplished by means of configuration source files.

2.6.2 Configuration Source Files

Each multi-transputer implementation requires a configuration source file. These files, which have the extension *cfs*, can be compiled and used with the linked unit process binaries to produce an executable file for the entire implementation. Each configuration source file contains C-like code that describes both the implementation's hardware (transputer) and software (linked unit process) configurations and the mapping between them. First of all, a description of the current transputer topology is given. This includes the types of all the transputers used and the links that are connected between them. Next, all the linked unit processes in the implementation and their parameters are given. The parameters are listed in a special *interface* statement after the name of the linked unit. Input and output channels can be among these parameters. These roughly correspond to transputer links and can be connected together in the configuration source file. The memory requirements of each linked unit must also be given. Optionally, a memory ordering and the execution priority of the unit can be added. Lastly, a process-to-processor mapping description is given. This simply states which linked units run on which transputers. In the general case, L linked unit processes can be placed on P transputers. The only restriction is that two connected processes cannot be placed on different transputers if there is no available connected link between the transputers for the processes to use for communication.

2.6.3 Program Compilation and Execution Processes

The compilation process for Parallel C programs is quite complicated. The following is a simplified version of what occurs. First of all, each source (*c*) file must be compiled using the *icc* command. The resulting transputer object code files are given

the extension *tco*. Next, the linker program *ilink* resolves all external references in these object files. It links the object files with the necessary libraries and produces a single linked unit for each with extension *lku*. In the case of multi-transputer programs, the network configurer *icconf* then compiles the configuration source file of the implementation into a binary configuration data file having extension *cfb*. In the last stage of compilation the implementation code is collected into a single bootable binary file with extension *btl*. This is the job of *icollect*, the code collector. For single transputer programs, *icollect* generates a bootable file by simply adding bootstrap code to the single linked unit. However, in the case of multi-transputer programs, the collector uses the information in the configuration data file to combine all linked units into a bootable. In either case, the bootable contains all transputer executable code and information for passing it to the transputers in the network. The *iserver* writes the bootable file to the boot link of the root transputer and execution begins. In Figure 2.3 there is a flow chart diagram of the compilation and execution processes.

2.7 Remarks

When complemented with a toolkit supplied by INMOS, the B008 containing T800 transputers programmed with Parallel C forms a complete parallel programming environment. Its greatest advantage is its flexibility. Its MIMD architecture dictates that both data parallel and instructional parallel algorithms can be implemented. Another positive characteristic is its relative inexpensiveness. Its major disadvantage, on the other hand, is that a great deal of training and background knowledge is required to learn how to use it properly. Multi-transputer programming is a challenging, complicated exercise even to an experienced sequential programmer. Once mastered, however, it can be used to efficiently program a wide variety of applications.

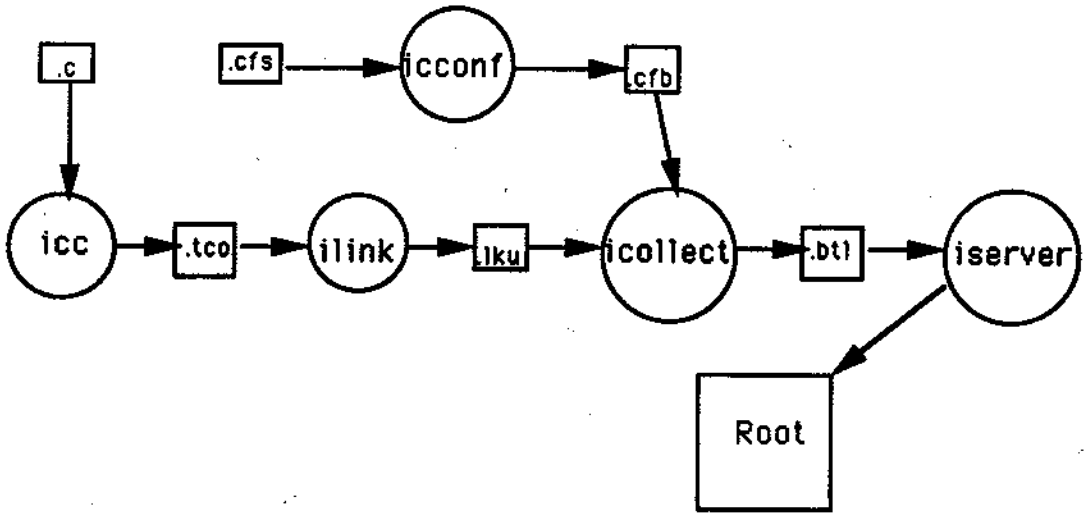

Figure 2.3: Parallel C Compilation Process

Chapter 3

Image Processing

3.1 Introduction

In this chapter we introduce image processing, concentrating as in Chapter 2 on the areas needed to understand this thesis. Image processing has been undertaken for nearly three quarters of a century. Computers were not used extensively for the task before the 1960's despite the fact that their ability to execute instructions quickly on a large amount of data made them well suited to it [9]. Image processing encompasses areas such as image enhancement and pattern recognition. Image enhancement involves the improvement of the quality of images for human observation. Its techniques include histogram equalization, crispening and smoothing. Pattern recognition, on the other hand, attempts to place visual patterns into one of many possible classes. To do so, distinguishing features or measures are first extracted from each pattern. These features are used to determine which class the pattern best fits. The features can be either local or global. Our work focuses on the two feature extraction techniques of convolution and statistical moments.

The chapter proceeds as follows. First we describe how images are represented in computers. Image processing equipment is discussed in the next section. A section about image processing systems comes next. We devote one section to each of the

two feature extraction techniques implemented in our work, convolution and regular moment extraction. Some remarks about the algorithms used in our work conclude the chapter.

3.2 Digital Representation of Images

Images must be converted to digital form for storage and processing on computers. This digitization occurs in two ways, spatially and in brightness. First, images must be sampled at a finite number of points in space. Usually, this sampling is done on square $N \times N$ grids where N is equivalent to a power of 2. Each sampled point is referred to as an image pixel. The pixels are assigned a discrete intensity value called a grey level which approximates the image brightness at that point. These values are usually non-negative integers that range from 0 to $I - 1$ with I again usually being equal to a power of 2. The resolution of a digital image refers to the number of pixels it has. The greater the number of pixels, the higher the resolution will be.

The resulting digital representation of the image is referred to as a grey level image. Clearly, the greater the values of N and I, the more accurate the grey level image will be. However, as N and I increase, the number of bits required to store the image increases as $N^2 \log_2 I$. In particular, the storage requirements for a 1024×1024 image with 256 grey levels would be $1024 \times 1024 \times 8$ bits or 1 Megabyte. This is acceptable for storage on disk, but might be impossible for main memory storage in smaller computers. More importantly, the processing time needed for most image processing techniques increases quadratically with the image size N. This may entail that the system response time is too slow on some computers for large images. It is also important to note that images with a relatively narrow dynamic range in intensity values may not require a large dynamic range of grey levels. Figure 3.1 shows a 256×256 grey level image with 256 original grey levels. (The printed image has been enlarged and may contain fewer grey levels.)

Figure 3.1: A Grey Level Image

3.3 Image Processing Equipment

3.3.1 Cameras

Cameras convert the visual scene before them into an analogue TV signal such as the RS-170. They use a two dimensional array of sensors which measure the brightness of each part of the scene to produce frames of analogue voltage pulses. The frames are stored in the camera's frame grabber. They are updated at least thirty times a second. The information in the frames is modulated onto an analogue television signal which is the output of the camera. Cameras that directly produce digital grey level images also exist. (See for example [22].) While usually quite expensive, they serve to eliminate the need for the analogue to digital conversion of video signals usually performed by digitizers.

3.3.2 Digitizers

Digitizers convert the analogue voltage signals of image frames produced by most cameras to grey level intensity values. Quantization errors (the difference between the grey level and the actual brightness) usually occur in the process.

3.3.3 Monitors

Monitors are used to display video images. They take an analogue video signal and output it to a raster screen.

3.4 Image Processing Systems

3.4.1 Need for Parallel Processing

Cameras usually produce at least thirty frames every second. Each frame in turn consists of thousands or even hundreds of thousands of pixels. Most image processing techniques require operations to be performed on each of the pixels. For this reason, real-time image processing is too computationally intensive for a standard, single processor computer. The use of parallel processing seems to be the only way to achieve real-time image processing.

3.4.2 Our Multi-Transputer Based System

An IBM PS/2 Model 30 286 is the host computer for our image processing system. A CCD (Charge Coupled Device) camera is used to acquire images. It is connected to the MATROX PIP 1024 video digitizer board [20, 15] located in the host. The PIP is used to instantaneously store a frame from the camera in digital form. The B008 motherboard is also inside the host. It is connected to the host's 16 bit AT bus by way of the C012 communication device. The host file system is used to store images.

A CHESTER monitor (not to be confused with the host monitor) also connects with the host and is used to display images.

The ALHAZEN Image Acquisition System, a program running on the host, gets images from the digitizer and writes them to the host file system. Parallel C programs executing on the B008 acquire these images from the resulting binary, 8-bit ASCII files stored on the host. The programs then perform an image processing task on the images and output the result to files on the host. ALHAZEN can also be used to display images on the monitor, both those stored in the PIP digitizer and those in files on the host. The S708B device driver runs on the host. It controls the transfer of data from the host to the root transputer of the B008. Our multi-transputer based image processing system is shown in Figure 3.2.

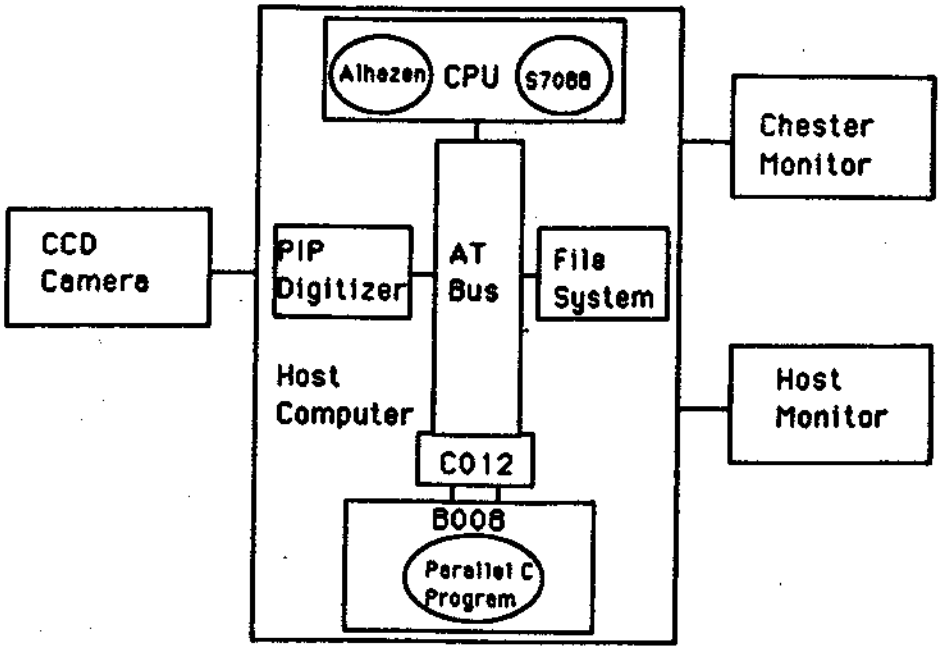

Figure 3.2: Our Multi-Transputer Based Image Processing System

3.5 Convolution

Convolution has been extensively used in many areas of image processing. It involves the passing of a mask or filter over the image. The features enhanced by convolution are local ones. This means that they exist independent of the rest of the image. Different types of masks can be used to extract different kinds of features. Some image features extractable using convolution are lines, edges and points. Two commonly used masks are the Laplacian and the Sobel. The convolution of two discrete two dimensional functions f and g is defined to be

$$f_e(x,y) * g_e(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f_e(m,n) g_e(x-m, y-n) \tag{3.1}$$

for $x = 0..M-1$ and $y = 0..N-1$, where f_e and g_e are extended versions of the functions f and g, with additional zeros added outside the domain of the functions so that the convolution operation can be performed.

The reader can verify by examining Equation 3.1 that convolution requires many multiplications and additions to be performed. For real-time applications, the procedure may be computationally too intensive for a normal single processor computer. One approach [3] involves using the Fast Fourier Transform (FFT) to compute the Fourier transforms of f_e and g_e, multiplying the transformed functions and taking the inverse transform of the result to get the convolved function. Our approach to the problem is to spread the computation required over many processors.

3.5.1 Algorithm

The convolution algorithm involves many multiplications and additions. It is shown in Figure 3.3. F is a two dimensional $N_1 \times N_2$ matrix storing the intensity function of the image. G is an $M \times M$ matrix storing the function of the mask. C is the two dimensional $N_1 \times N_2$ array storing the values of the convolved image function. It is assumed that $floor M/2$ zero pixels have been added around the edge of F.

```
CONVOL(F, N1, N2, G, M, C)
  for i = 0..N1 − 1
    for j = 0..N2 − 1
      C(i, j) = 0
      for k = 0..M − 1
        for l = 0..M − 1
          C(i, j)+ = F(i + k, j + l)G(k, l)
        end for
      end for
    end for
  end for
END
```

Figure 3.3: Convolution Algorithm

3.6 Moments

Statistical moments have also been widely used in image analysis. They are among the most significant of global image features. Many important geometric attributes of an image can be determined from its moments. Among these are its mass, spread and centre of inertia. Papoulis' uniqueness theorem [18] has established that no two images can have the same set of moments. This property ensures that moments are effective as a means of distinguishing among patterns. In addition, object features called moment invariants, which do not change if a geometric operation (as defined in Section 3.6.1) is performed on the object, can be derived from moments. Moments are therefore a valuable tool for image analysis. The formula for calculating m_{KL}, the (K, L)th regular moment of a discrete two dimensional function $f(i,j), i = 1..N, j = 1..N$ is

$$m_{KL} = \sum_{i=1}^{N} \sum_{j=1}^{N} i^K j^L f(i,j) \tag{3.2}$$

Central moments have also been frequently used in image processing. They can be derived from the regular moments of an image or can be calculated directly using

the formula

$$\mu_{KL} = \sum_{i=1}^{N}\sum_{j=1}^{N}(i-\overline{x})^K(j-\overline{y})^L f(i,j) \tag{3.3}$$

with

$$\overline{x} = \frac{m_{10}}{m_{00}}, \qquad \overline{y} = \frac{m_{01}}{m_{00}}.$$

From Equation 3.2 it can be seen that moment calculation also requires a large number of multiplications and additions. This entails that it is a slow process on most sequential computers. There are at least two approaches to resolving this dilemma. The first is to treat the image as a representation of a single object and to calculate the moments of its boundary. Research indicates that much of the information encoded in an object's moments originates in the boundary areas of the object. It is therefore legitimate to calculate moment information only about the object boundary. Calculation of the moments of an object boundary is usually less computationally intensive than calculation of an image's overall moments. Images are treated as binary digital representations of a single object. The object's boundary is determined. To calculate the moments of the boundary, a special mathematical or geometrical technique is usually used. For example, Leu [16] first splits the object into triangles. The moments of the triangles are then calculated. The overall object moments are then calculated using the triangle moments. Another method is used by Li and Shen [14]. They propose using Green's theorem to calculate the moments along the boundary of the object. The other approach to moment calculation is to find clever parallel algorithms to calculate the overall image moments on many processors. This is our approach in this thesis.

3.6.1 Moment Invariants

An object feature is said to be invariant if it does not change when the object undergoes at least one of the geometric operations of scaling, rotation, reflection or

translation. In 1961, Hu [12] proposed a set of invariant features based on moments. He derived six moment invariants using algebraic invariants from regular moments. They have been used in experiments to recognize planes [8] and ships [5]. In general, moment invariants are derivable from the regular moments of an image. In addition to their computational intensity, they have the drawback of being relatively noise sensitive. Many other moment invariants have been suggested besides Hu's. They have varying degrees of discrimination ability and noise sensitivity. Among these are Zernike moment invariants [24] and Bahmieh moment invariants [1]. For more information about moment invariants, the reader should consult the comparative study presented by Belkasim, Shridhar and Ahmadi [2].

3.6.2 Algorithms

There are eight algorithms for regular moment calculation used in our work. The most obvious way to calculate moments is the straightforward algorithm [7]. Figure 3.4 shows this algorithm for calculating the first 16 moments. In the moment algorithms, F is again the $N_1 \times N_2$ matrix containing the image intensity function. m is the 4×4 matrix containing the image moments.

Because $2N^2$ exponentiations are needed to compute each moment, this algorithm is quite inefficient. An algorithm that avoids these calculations should be better. Notice that the column coefficient monomials j^l can be calculated using j^{l-1}. In particular,

$$j^l = jj^{l-1}. \tag{3.4}$$

Reeves' coefficient storage algorithm [23] exploits this fact. The idea here is to create four $N_1 \times N_2$ arrays $C(l), l = 0..3$ of column coefficient monomials using equation 3.4. These are then used to calculate the moments. Figure 3.5 shows the coefficient storage algorithm for the calculation of an image's first 16 regular moments.

```
STRAIGHT(F, N1, N2, m)
  for k = 0..3
    for l = 0..3
    m_kl = 0
      for i = 0..N1 - 1
        for j = 0..N2 - 1
          m_kl+ = (i + 1)^k (j + 1)^l F(i, j)
        end for
      end for
    end for
  end for
END
```

Figure 3.4: Straightforward Algorithm for Moment Calculation

One drawback of the coefficient storage algorithm is that the products of image elements $F(i,j), i = 0..N_1-1, j = 0..N_2-1$ and coefficients are repeatedly calculated in the final stage of the algorithm. We can solve this problem by incorporating the image values in the four coefficient arrays. We call the resulting algorithm column product storage. The column product storage algorithm is shown in Figure 3.6.

There are two main problems with the storage algorithms. First, to process an $N \times N$ image, the storage arrays require $8N^2$ bytes, a great deal of memory for large values of N. In addition, $O(N^2)$ multiplications would be needed to execute the algorithms on such an image. Multiplications are relatively computationally expensive for small VLSI processors.

The double summation in Equation 3.2 can be split into two separate summations. Equation 3.5 gives the resulting equation for the calculation of m_{KL}.

$$m_{KL} = \sum_{i=1}^{N} i^K \sum_{j=1}^{N} j^L f(i,j) = \sum_{i=1}^{N} i^K RM_{Li} \tag{3.5}$$

where

$$RM_{Li} = \sum_{j=1}^{N} j^L f(i,j)$$

```
COE(F, N_1, N_2, m)
   for i = 0..N_1 - 1
       for j = 0..N_2 - 1
           C(0, i, j) = 1
       end for
       for l = 1..3
           for j = 0..N_2 - 1
               C(l, i, j) = (j + 1)C(l - 1, i, j)
           end for
       end for
   end for

   for l = 0..3
       for k = 0..3
           m_kl = 0
           for i = 0..N_1 - 1
               for j = 0..N_2 - 1
                   m_kl+ = C(l, i, j)F(i, j)
                   C(l, i, j)* = i + 1
               end for
           end for
       end for
   end for
END
```

Figure 3.5: Coefficient Storage Algorithm for Moment Calculation

```
COL(F, N_1, N_2, m)
   for i = 0..N_1 - 1
       for j = 0..N_2 - 1
           C(0, i, j) = F(i, j)
       end for
       for l = 1..3
           for j = 0..N_2 - 1
               C(l, i, j) = (j + 1)C(l - 1, i, j)
           end for
       end for
   end for

   for l = 0..3
       for k = 0..3
           m_kl = 0
           for i = 0..N_1 - 1
               for j = 0..N_2 - 1
                   m_kl+ = C(l, i, j)
                   C(l, i, j)* = i + 1
               end for
           end for
       end for
   end for
END
```

Figure 3.6: Column Product Storage Algorithm for Moment Calculation

The calculation of the (K, L)th moment is thus split into two steps. First, the Lth one dimensional moment RM_{Li} of each image row is calculated. These row moments are then used to calculate the (K, L)th overall two dimensional moment m_{KL}. This is not unlike using the one dimensional Fourier transform to calculate the transform of a two dimensional function. The next three algorithms all use this technique to solve the problems associated with the storage algorithms.

We have created an algorithm similar to the storage algorithms that uses the above two step moment calculation technique. We have called it the add and multiply algorithm because addition and multiplication alternate in the process of moment calculation. Figure 3.7 shows the one dimensional add and multiply algorithm for the calculation of the first four row moments of a two dimensional discrete $N_1 \times N_2$ function F. The moments are stored in M.

```
1DADDANDMU(F, N1, N2, offset, M)
   for i = 0..N1 − 1
      for l = 0..3
         M(l, i) = 0
         for j = 0..N2 − 1
            M(l, i)+ = F(i, j)
            F(i, j)* = (j + offset + 1)
         end for
      end for
   end for
END
```

Figure 3.7: One Dimensional Add and Multiply Algorithm for Moment Calculation

The two dimensional add and multiply algorithm for the calculation of the first sixteen overall image moments of an $N_1 \times N_2$ image with intensity function F is equivalent to

$$1DADDANDMU(F, N_1, N_2, 0, RM); 1DADDANDMU(RM, 4, N_1, 0, m)$$

Although it has relatively small memory requirements, this algorithm still requires

$N^2 + N$ multiplications to calculate each moment of an $N \times N$ image. It would be preferable that an algorithm use only additions to calculate moments. Budrikis and Hatamian have proposed an algorithm for regular moment calculation that does so [4, 10]. The algorithm uses recurrences to build up higher order moments from lower ones. To calculate the first 16 moments of an $N \times N$ image, it only requires $8N^2 + 32N$ additions. Figure 3.8 gives the pseudo code for this recursive algorithm. Again the first four one dimensional moments of the rows of a two dimensional $N_1 \times N_2$ image function F are calculated and stored in M.

```
1DRECURSE(F, N1, N2, M)
   for i = 0..N1 − 1
       M0 = M1 = M2 = M3 = 0
       PREVM1 = PREVM2 = 0
       for j = 0..N2 − 1
           M0+ = F(i, N2 − 1 − j)
           M1+ = M0
           M2+ = M1 + PREVM1
           M3+ = M2 + M2 + PREVM2 − M1
           PREVM1 = M1
           PREVM2 = M2
       end for
       M(0, i) = M0
       M(1, i) = M1
       M(2, i) = M2
       M(3, i) = M3
   end for
END
```

Figure 3.8: One Dimensional Recursive Algorithm for Moment Calculation

As is the case with the two dimensional add and multiply algorithm, the two dimensional recursive algorithm is equivalent to

$$1DRECURSE(F, N_1, N_2, RM); 1DRECURSE(RM, 4, N_1, m)$$

Chen [7] has proposed another regular moment extraction algorithm that uses

only additions. He calls it the partial sum algorithm. The algorithm also uses lower order moments to calculate higher order ones. The technique employed here is to build up higher order moments by repeatedly partially summing over lower order moments. To calculate each row moment, each pixel is added to the pixel before it. Next each pixel is added to the pixel two before it. This process continues with the distance between added pixels doubling at each iteration until the distance is equal to half the image width. In this way, $O(N^2 \log_2 N)$ additions are required to calculate each set of row moments of an $N \times N$ image except the last. They may be calculated by simply summing the previous values of the intensity function F. As with the previous two algorithms, overall moments can be calculated using two applications of this technique. The partial sum algorithm for the calculation of the first four row moments of a two dimensional $N_1 \times N_2$ function F is given in Figure 3.9.

In addition to the two dimensional add and multiply, recursive and partial sum overall moment extraction algorithms, six additional two dimensional algorithms can be created. To do so, the one dimensional algorithms are combined in pairs with one algorithm calculating the row moments and the other calculating the overall moments. Only two of these hybrid algorithms will be used in our research. Both use the recursive algorithm to calculate the row moments. The first uses the add and multiply algorithm to calculate the overall moments. The other uses the partial sum algorithm to do so. The first will be referred to as the recursive/add and multiply hybrid, the other as the recursive/partial sum hybrid.

3.7 Remarks

The convolution algorithm requires M^2N^2 floating point multiplications and additions. Especially for large mask sizes M, we anticipate that it will have a slow single transputer implementation. We also intend to implement the straightforward, coefficient storage, column product storage algorithms and the two dimensional add and

```
1DPARSUM(F, N1, N2, M)
   n = log2 N2
   for i = 0..N1 − 1
       for l = 0..2
           for index = 0..n − 1
               offset = 2^index
               for j = 0..N2 − 1 − offset
                   F(i, j)+ = F(i, j + offset)
               end for
           end for
           if l ≠ 2 then
              M(l, i) = F(i, 0)
           end if
       end for
       firstOrderMomentSum = F(i, 0)
       for j = 0..N2 − 2
           F(i, j)+ = F(i, j + 1)
       end for
       M(2, i) = F(i, 0)
       for j = 1..N2 − 1
           F(i, 0)+ = F(i, j)
       end for
       M(3, i) = 3F(i, 0) + M(2, i) − firstOrderMomentSum
   end for
END
```

Figure 3.9: One Dimensional Partial Sum Algorithm for Moment Calculation

multiply, recursive, and partial sum algorithms on a single transputer. Of these six algorithms, only the recursive, partial sum, and add and multiply algorithms seem to hold much promise. The straightforward algorithm requires too many exponentiations to be efficient. The memory requirements of the two storage algorithms are prohibitive. Two hybrid algorithms will also be implemented for analysis purposes, as has been stated.

Chapter 4

Single Transputer Implementations

4.1 Introduction

Due to the large number of single processor algorithms used in our work, the single transputer implementations are quite extensive. In particular, nine of the thirteen single process algorithms mentioned in Chapter 3 have been implemented on a single transputer using Parallel C. Our goals in doing so are as follows: 1) to gain familiarity with developing programs in Parallel C on transputers, including, among other things, learning about the compilation process, creating makefiles and configuring source programs to run on transputers; 2) in the case of regular moments, to find the fastest algorithms for parallelization; 3) to create base code from which we could develop the multi-transputer implementations later; and 4) to develop straightforward correct implementations in order to assure ourselves that those more complicated multi-transputer implementations are correct.

In this chapter, we first describe the technique we have used to program the single transputer. A section devoted to the single transputer convolution implementation comes next. Timing results and analysis for the eight regular moment programs are

presented next. Some remarks conclude the chapter.

4.2 Programming Technique

The single transputer programs have been compiled and executed on the root T800 transputer of the B008 motherboard. All single transputer programs have been implemented as Parallel C programs which use none of the parallel features of the language. In particular, no processes are allocated or run, no channels are used and no message passing is performed. With only minor modifications, these programs could be compiled and run on a standard computer architecture with a standard C compiler such as a Sun 4 running ANSI C under UNIX. Each program performs its computation on the image and outputs the result (a convolved binary image in the case of convolution, sixteen moments in the case of regular moments) to files on the host. The time taken for computation alone is then printed on the host monitor.

4.3 Convolution

The convolution algorithm presented in Section 3.5.1 has been implemented on a single transputer using Parallel C. Both the image and mask are acquired from files on the host. The image pixels are stored as single byte Parallel C *char* data types. The mask elements are stored as *float* data types. In Parallel C these are implemented as 32 bit IEEE floating point numbers. To maintain full flexibility for convolving an arbitrary sized image and with an arbitrary sized mask, the mask and image are stored in dynamically allocated buffers rather than in static Parallel C arrays. Offsets into these buffers are maintained in order to reduce the time needed to reference elements.

Due to the large number of multiplications and additions needed, the convolution algorithm runs relatively slowly. The program has been run on images with dimensions equal to powers of two from 64 to 1024 for four different sized real masks. Table

4.1 shows all the execution times in seconds. N represents the image size, M the mask size.

M \ N	64	128	256	512	1024
3	0.3239	1.293	5.166	20.65	82.61
5	0.7338	2.934	11.73	46.93	187.7
7	1.33	5.325	21.31	85.24	341
13	4.237	16.99	68.02	272.2	1098

Table 4.1: Single Transputer Convolution Timing Results in Seconds

Figure 4.1 gives a plot of the execution time against the number of image pixels for the four different mask sizes. The execution time increases linearly with the number of image pixels (quadratically with image size N). A graph of the convolution execution times as the mask size increases for the five different sized images is given in Figure 4.2. The reader can verify that the times increase slightly less than linearly with the number of mask elements by examining Table 4.1. This is due to the fact that the multiplications and additions involved in convolution must be performed not only on the image pixels but also on the border of zeros added around the image. The arithmetic operations on zero element values seem to be slightly faster than arithmetic with any other value in Parallel C on the T800 transputer. Furthermore, the number of additions required to maintain the buffer offsets increases less than linearly with the number of mask elements. We have modified the convolution program to accept integer masks. Not surprisingly, the execution times are faster than those achieved with floating point masks, since integer arithmetic operations are faster than floating point ones.

4.4 Regular Moments

The six pure and two of the hybrid moment extraction algorithms described in Section 3.6.2 have also been implemented on a single transputer. As is the case with the convolution implementation, the images are acquired from host files and stored

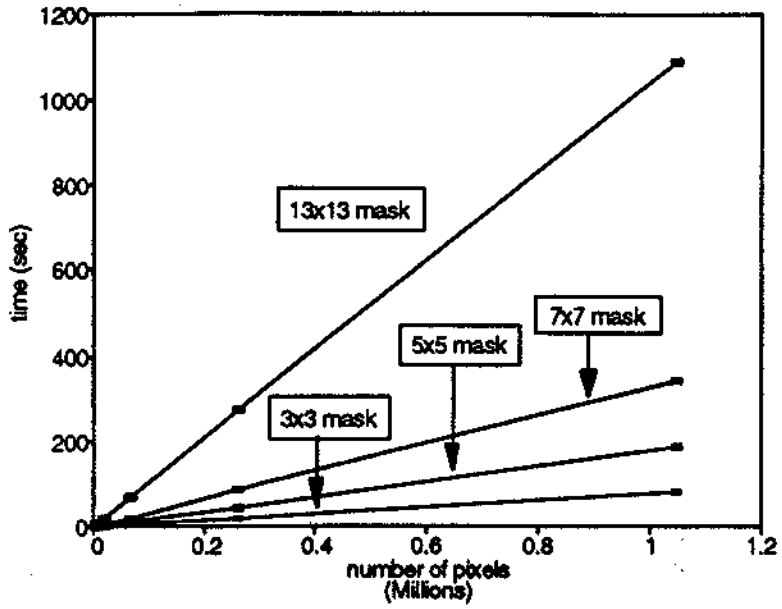

Figure 4.1: Convolution Timing Results for Various Mask Sizes

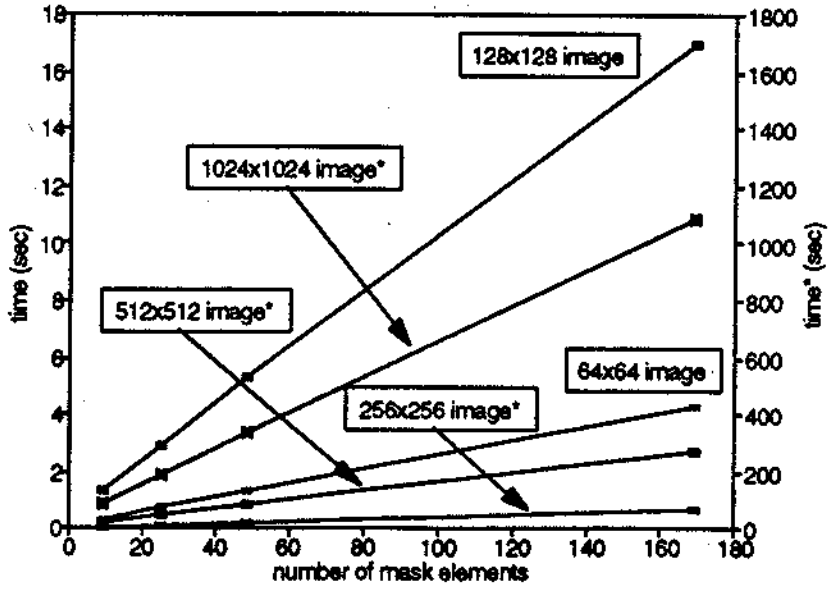

Figure 4.2: Convolution Timing Results for Various Image Sizes

in buffers of Parallel C *char* types. Offsets into these buffers are again maintained during execution of the programs. The first sixteen moments are found by all the implementations. They are calculated as Parallel C *double* types. These are implemented as 64 bit IEEE double precision floating point numbers. The moments are printed to a host file at the end of each implementation. Each program was run on the same five image sizes as the convolution program. Table 4.2 shows the timing results in seconds. N is the image size. ISM indicates there was insufficient memory to run the program on the image size in question.

algorithm / N	64	128	256	512	1024
straightforward	21.27	85.5	342.8	1373	5495
coefficient	1.069	4.26	17.01	ISM	ISM
column products	0.7935	3.158	12.6	ISM	ISM
recursive	0.05888	0.2249	0.8803	3.484	13.86
add and multiply	0.1648	0.6634	2.484	9.839	39.17
partial sum	0.4648	2.08	9.322	41.57	184
recurse/multiply	0.06522	2.375	0.9055	3.534	13.96
recurse/partial	0.08723	0.2921	1.036	3.839	14.66

Table 4.2: Single Transputer Regular Moment Timing Results in Seconds

Figure 4.3 contains a graph of the execution times of the straightforward algorithm plotted against the number of image pixels. The execution time increases approximately linearly with the number of image pixels. Due to the large number of additions, multiplications and exponent calculations required, this is by far the slowest of the single transputer implementations.

Similar graphs of execution times versus number of image pixels for the coefficient storage and column product programs are given in Figure 4.4. Again, linear increases in the execution times with the number of pixels are shown. There was insufficient memory for these two programs to be run on images greater than 256×256. The timing results for the two dimensional partial sum, add and multiply and recursive algorithm implementations are plotted in Figure 4.5. The times for the recursive and the add and multiply implementations increase slightly less than linearly with the

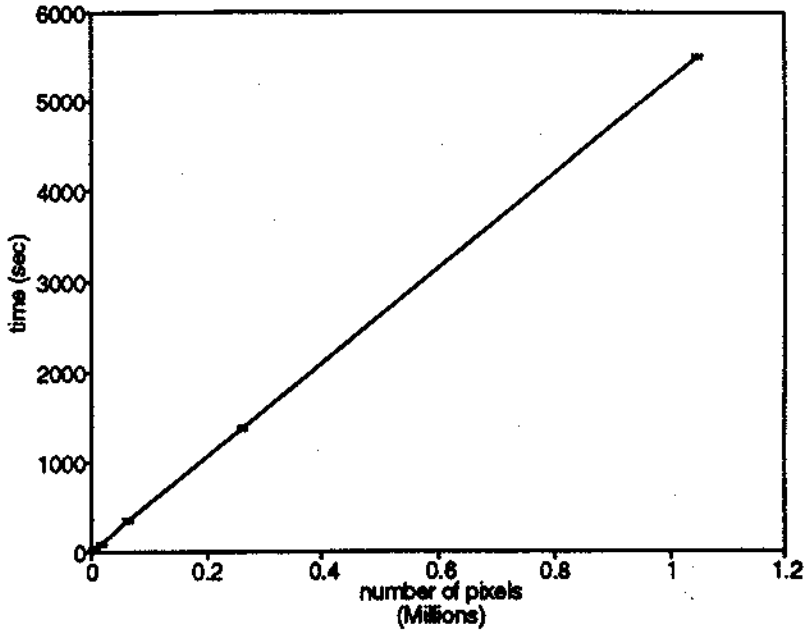

Figure 4.3: Straightforward Algorithm Timing Results

number of image pixels. However, the execution time of the partial sum implementation increases more than linearly with the number of pixels. The reader can confirm these observations by referring to Table 4.2. These three programs are in fact the fastest single transputer moment implementations.

In order to allow speedup and efficiency analyses to be performed in Chapter 5 on two of the multi-transputer programs, the recursive/partial sum and recursive/add and multiply hybrid algorithms have also been implemented. Figure 4.6 contains graphs of the execution times of these two single transputer hybrid programs. The execution times here increase slightly less than linearly. This can confirmed by referring to Table 4.2.

We believe that the timing results of the four hybrid algorithms not implemented would fall between those of the two pure programs involved. A final observation is that when the data type of the moments is changed from double to single precision

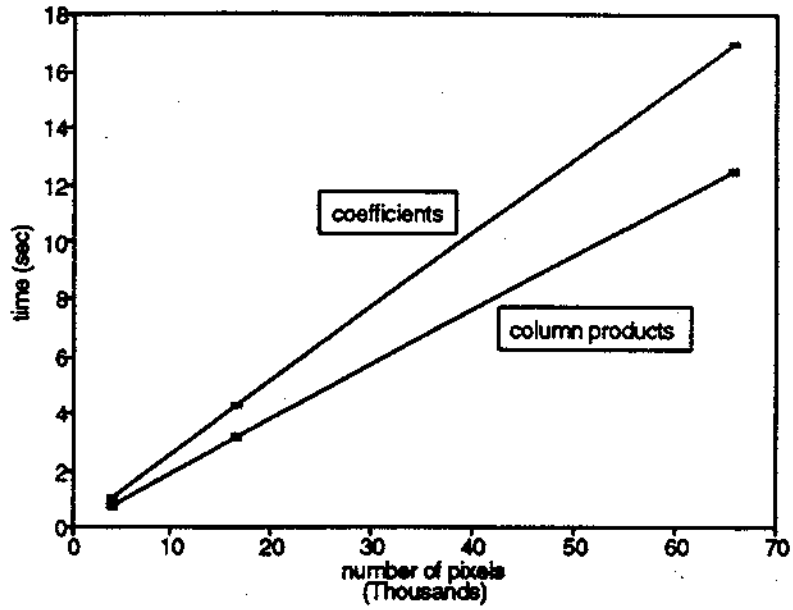

Figure 4.4: Coefficient and Column Product Algorithm Timing Results

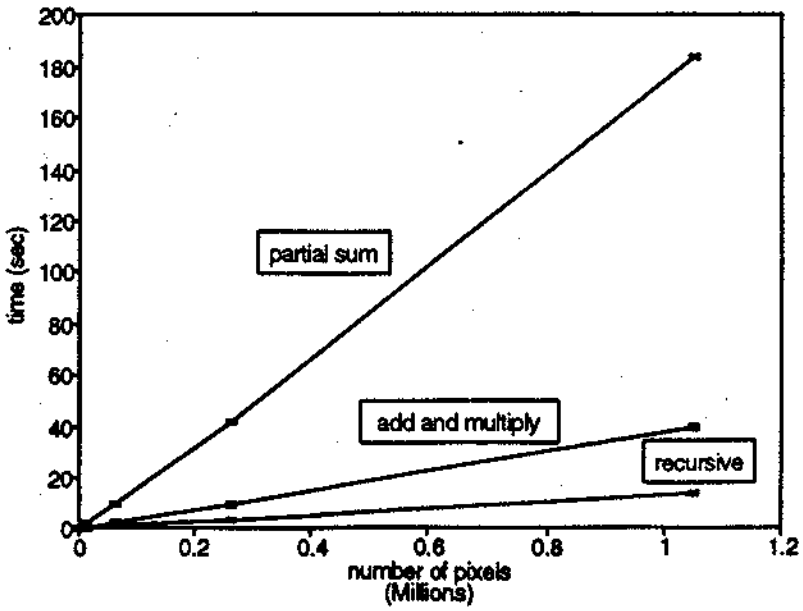

Figure 4.5: Recursive, Partial Sum and Add and Multiply Algorithm Timing Results

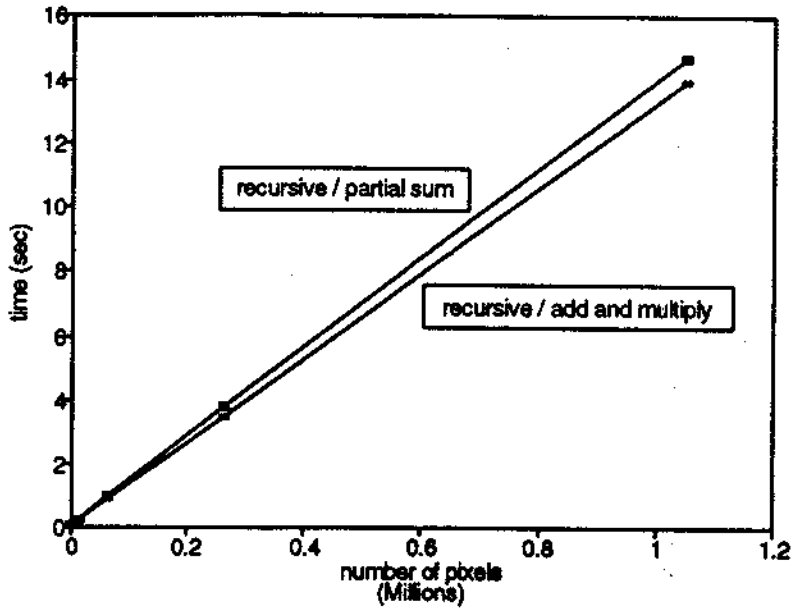

Figure 4.6: Hybrid Algorithm Timing Results

in the recursive program, a significant decrease in execution times results. We expect that similar reductions would occur if single precision moments were calculated in the other programs. Some loss of accuracy in the moments extracted results when this is done, however.

4.5 Remarks

All the single transputer programs were compiled using *icconf* and a configuration source file. Alternatively, they can be compiled without a configuration source file using the single transputer option of *icollect*. (See Section 2.6.3 for a description of the transputer compilation procedure.) The latter method produces significantly faster running programs. We believe that is this due to the fact that less general code is produced if no parallel programming features are needed. Our method of compilation

allows fairly accurate multi-transputer models to be presented in Chapter 6 since the multi-transputer programs need to be compiled in the most general parallel way. The single transputer implementations themselves are quite extensive. However, we believe the time spent programming them was worthwhile since they provide a good foundation for the multi-transputer programs presented next.

Chapter 5

Multi-Transputer Implementations

5.1 Introduction

Four multi-transputer implementations of both convolution and regular moments have been realized. Our goals in doing so are loftier than they are in the single transputer case. Certainly we wish to create fast programs for performing convolution and for calculating regular moments on the B008 using Parallel C. In the process, however, we also wish to determine and investigate the issues related to image processing on transputers in general. In particular, we want to discover useful principles and to create strategies for implementing more general image processing applications efficiently on multiple transputers.

This chapter begins with a description of the programming technique we have used in the multi-transputer case. Next, descriptions of the issues relevant to image processing on transputers and our strategies for dealing with them are presented. A section describing the multi-transputer implementations for convolution is followed by a similar section that gives the details of the regular moment multi-transputer implementations. Analysis of the implementations is given next. As usual, the chapter

ends with some concluding remarks.

5.2 Programming Technique

The entire B008 motherboard is used in the multi-transputer implementations. (See Section 2.5.7 for a full description of the B008 and its operation.) Our method of programming here is quite different from that used in the single transputer implementations. In the single transputer case, a Parallel C program running on the root transputer is not only responsible for controlling input and output to the host but also for all computations needed to perform the image processing tasks. The other transputers are not used. However, in our multi-transputer implementations, all nine transputers may be used. Parallel C is again used. In this case, however, one separate asynchronous linked unit process executes on each transputer used. The root transputer is still responsible for data transfers to and from the host. All of the necessary computations are executed on the other eight transputers.

Each transputer has been programmed using the optimal transputer programming method discussed in Section 2.5.9. All local channels and processes are declared as global variables in the source file for each transputer. The *main()* function of each source file is used to allocate channels, allocate and run processes and to time the implementation. Processes are run asynchronously in parallel using the *ProcRun()* function. A channel is created for communication between each pair of processes that communicate in each direction they communicate. Aside from channels and processes, global data structures have been kept to a minimum. In this way, the problems of mutual exclusion are avoided that usually occur when more than one concurrent process accesses the same data. As in Hoare's model, each process keeps its own copy of data. This data can be sent to another process using message passing on a local channel. Configuration source files (see Section 2.6.2) are used to map the linked units to the appropriate transputer on the B008 for execution.

5.3 General Issues and Strategies

When we began this part of the research, we were confronted with many interesting and challenging questions. Among these were: "What parallelization techniques should be used?", "How should data be passed?", "What moment algorithm should be chosen?" and "What topologies are best for image processing?". For each question, a number of solutions are possible. One approach that we considered was simply to select one possible solution to each problem and to design and program an implementation. We could find the best implementations by simply repeating that process for all reasonable possibilities.

To have gone through with the above plan would have, of course, been foolhardy. For one thing, we would have been obliged to consider a bewildering number of implementations. Given the possible existence of other overlooked implementations, an equally serious defect with the idea is that we would have had no theoretical justification for claiming that we had indeed found the best implementations. All things considered, we decided on the contrary that we needed to examine each relevant issue separately and use theoretical reasoning in order to obtain the best strategy or strategies to deal with it. We could then design our implementations using those strategies. Using this method drastically reduced the number of implementations we had to consider. We also feel that this method gives us some scientific justification for believing that our implementations are superior. Indeed, we believe this approach has led to the design of efficient implementations.

Various parallel techniques have been discussed in Section 2.3. Data parallelism is one of the main techniques described. Under this scheme, many processors execute the same sequence of instructions on separate parts of a large amount of data. Image processing techniques generally require that the same operation be performed on a large number of pixels. For this reason, we have decided to focus on data parallel methods in our implementations. However, we realize that we might also have to

use more complicated instructional parallel techniques as well. In the upcoming sections we describe the following issues specific to multi-transputer image processing: initial data pass, computational algorithm selection, accumulation of final results and topology selection. For each issue, we relate the strategy we have decided upon to deal with it, and give our justification for that decision.

5.3.1 Initial Data Pass

Under the data parallel model, each transputer is responsible for processing a part of the image. These image parts might be passed directly to the transputers using the direct link access scheme suggested by Kille, Ahlers and Schneider [13]. In the direct link access system, one link of each transputer is permanently connected to a specially designed frame grabber transputer which directly passes each transputer its part of the image. Because these links cannot be used to configure the network, direct link access reduces the flexibility of the image processing system. Alternatively, the links of the frame grabber might be connected into the crossbar. The other transputers' links could then be used for network reconfiguration as usual. Our approach is closest to the second option.

The B008 root transputer acquires the images from files on the host. With a view to having a balanced computational load on the network and a symmetric computational model, we decided that the root transputer would not be used for computational purposes. It is used as a sort of frame grabber transputer that gets images from the host and passes computational results back. It is also responsible for passing the image segments to the computational transputers for processing. There are two main issues to resolve about this initial data pass. The first is how to split up the image. The second issue is which algorithm to use to pass the image segments to the transputers.

Splitting the Image

There are three options for splitting the image. It can be done row by row, column by column or section by section. Convolution can be done equally well on image rows, columns or sections. However, regular moments can be calculated by independently calculating row moments or column moments, and then using these moments to calculate the overall image moments (see Section 3.6.2). This indicates a row or column oriented distribution scheme. However, ALHAZEN stores its images row by row, not column by column. This last factor entails row by row transfer is the best way to pass the data. Therefore, in our implementations each computational transputer is passed and is responsible for processing several rows of the image.

Data Passing Algorithm Selection

There are at least two general algorithms for passing the image rows. The first might be called 'grouped'. In the grouped approach, the root simultaneously passes all the rows needed down each network path all at once, and every transputer on a path keeps only the rows for which it is responsible, passing the remainder down the path. The second might be called 'packet'. Under this scheme, the root simultaneously passes the rows needed down each path as separate packets, one after another. The packet for the last transputer in line is passed first, the packet for the second last second and so on until the packet for the first transputer is passed last. Each transputer passes all the packets needed further down the path and then receives and keeps its own packet.

For the purpose of passing the image rows, two subtopologies are used in our implementations. The first one is the pipeline structure shown in Figure 5.1. In this simple topology, the root is connected to one or more transputers in a chain. Timing results for both data passing algorithms for a pipeline with four transputers besides the root are given in Figure 5.2. The image pixels are distributed evenly among the

four transputers. There is insufficient memory to use the grouped method to pass the largest image. Clearly the packet algorithm is superior.

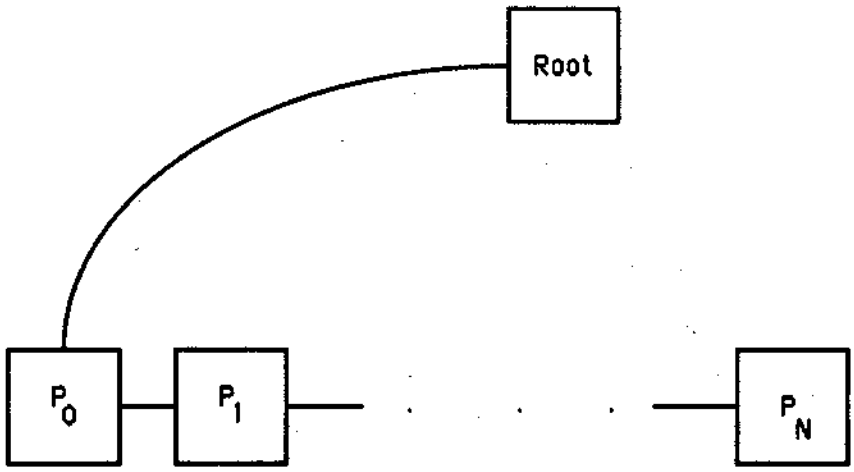

Figure 5.1: Pipeline Subtopology

The second subtopology is treelike, as shown in Figure 5.3. Here, the root is connected to a router transputer which in turn is connected to three more transputers. The timing results for the grouped and packet algorithms for this topology are given in Figure 5.4. The image pixels are again distributed evenly among the four transputers. The packet method is once again superior.

Based on the results in Figure 5.2 and in Figure 5.4, we have chosen the packet method as the initial data passing algorithm for our implementations. An additional advantage to using the packet algorithm is that larger images can be processed with it than can be processed with the grouped algorithm.

Clearly other algorithms exist for the initial data pass. In particular, the above two basic algorithms can be combined for some networks. For example, in a network

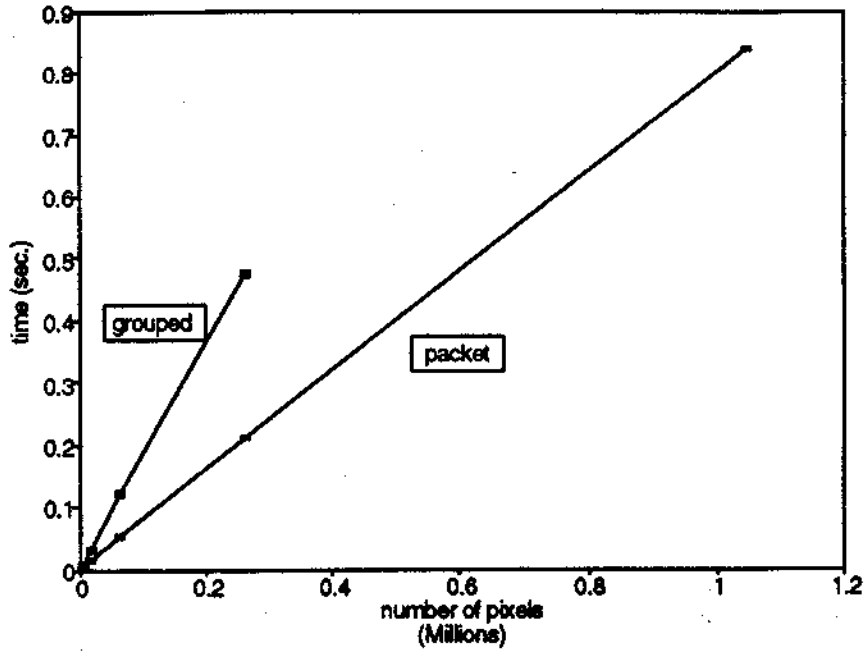

Figure 5.2: Pipeline Data Passing Timing Results

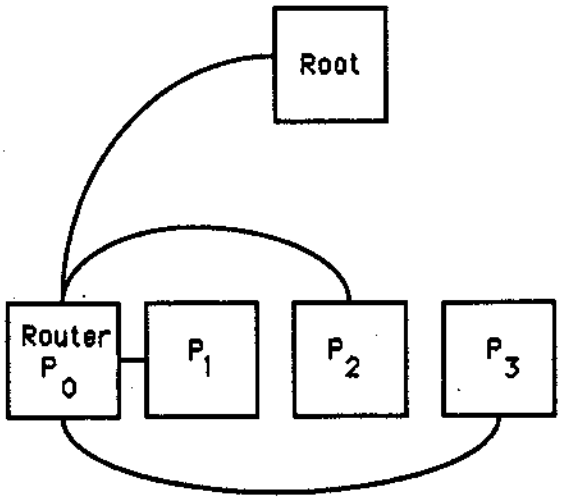

Figure 5.3: Tree Subtopology

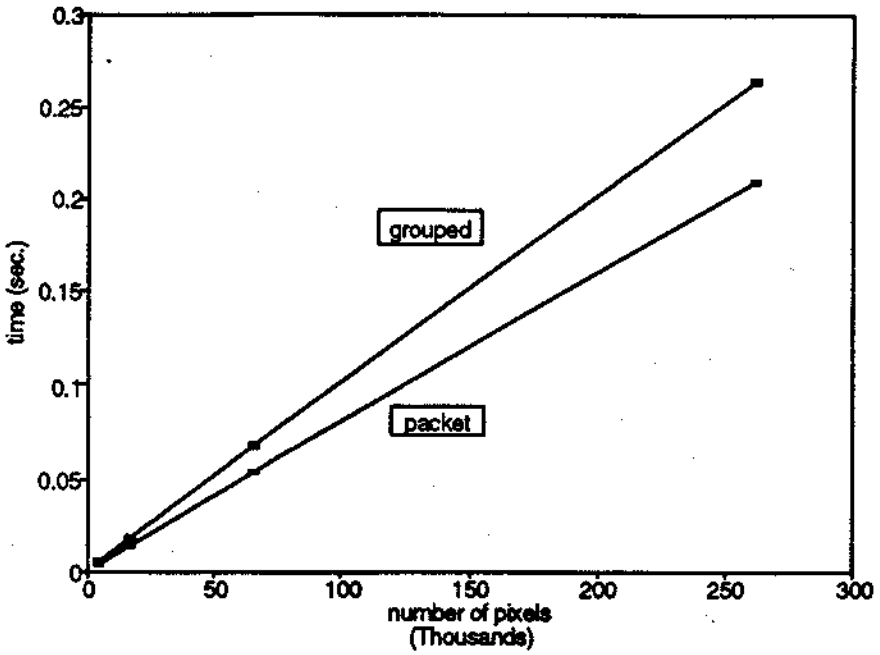

Figure 5.4: Tree Data Passing Timing Results

using the second subtopology the rows for the three leaf transputers might be passed as a group followed by the rows for the router. However, it is our conjecture that the packet method is the optimal algorithm for the initial data pass for all topologies in all networks. The fact that INMOS has chosen a packet scheme for its virtual channel data passing system on the T9000 gives some support to this hypothesis.

5.3.2 Computational Algorithm Selection

There are two ways to design parallel computational algorithms. One can either base them on existing single processor algorithms or create new parallel algorithms which have no relation to any known single processor algorithm. We have chosen the former technique, since it is much simpler. In particular, we have considered both data and instructional parallel versions of the sequential algorithms found in Chapter 3 and combinations of both. This has enabled us to convert actual code from the single

transputer implementations for use in these multi-transputer programs.

5.3.3 Accumulation of Final Results

Sometimes the computational stage of a multi-transputer program ends with the final results either in the root transputer or in the one connected to it. Also, in the direct link access model described in Section 5.3.1, the final results can be passed back to the frame grabber (root) directly. In other situations however, the final results must be passed back through the network to the root for output on the host. This process is the inverse of the initial data pass. There are again various ways to execute this data pass. Each one has an analogue initial data pass algorithm. For this reason, we have decided to use the same packet algorithm for the final data pass. In particular, each computational transputer passes its own results back towards the root and then passes back the results of those farther down its path. The root simultaneously gathers the packets from each network path ending at it.

5.3.4 Topology Selection

If we had chosen to use transputers to determine the best parallel implementations of moments and convolution for common parallel processing topologies, then we clearly would have been obliged to choose a topology first and then design the data passing and computational algorithms of the implementation with that topology in mind. However, we decided to find the best implementations for transputers themselves. This gives us freedom to choose the best algorithms independent of topology. We choose topologies only after algorithm selection as a way of enabling and optimizing a potential implementation. In particular, we choose topologies to make possible all necessary data communication and to minimize the time taken for it. This data communication can be done for the initial data pass, computational data communication or for the accumulation of final results.

In order to minimize the time taken for data communication a topology must be selected that has the shortest possible paths between transputers which communicate a lot of data or which communicate frequently. This is not always easy to accomplish, since sometimes many transputers must communicate a great deal and there are only four communication links available on each transputer. In addition, the optimal topology for each of the three communication steps may not be the same and a compromise topology may have to be created. Fortunately, this was never necessary in our work because the best topology for each of the steps was always the same. Because of this and because it is theoretically possible to reconfigure transputer networks dynamically we have decided that the issue of designing compromise topologies is not worth investigating in this thesis.

5.4 Convolution

In this section, we describe the multi-transputer programs we have created for performing convolution. The purely local nature of convolution entailed that it was less challenging a process than regular moments to implement on transputers. However, there were still issues about the initial data pass and topology selection to be investigated. In the next section we describe the reasoning that led to the design of the convolution implementations. Detailed descriptions and timing results for the implementations follow.

5.4.1 Design of the Implementations

Keeping in mind the general strategies described in Section 5.3, we have designed and implemented four multi-transputer convolution programs. We have created pure data parallel programs with each computational transputer receiving and processing several rows of the image. In each implementation, the frame grabber root transputer uses the packet algorithm to pass the image rows and the mask. The same convolution

algorithm has been implemented in all four programs as a data parallel computational process on all the computational transputers used. The packet method is again used to pass the rows of the convolved image back to the root in all the programs. Two pipeline convolution implementations investigate an issue related to the initial data pass. We have investigated the issue of topology selection by implementing ring and tree versions of one of the two pipeline programs. As in the single transputer case, all the programs use a mask consisting of floating point values and store the image pixels in buffers of *char* data types.

5.4.2 Pipeline Implementations

In the single transputer version of convolution, the image is first padded with a border of zero pixels. In the multi-transputer programs, the borders can either be added in the computational transputers once the image rows have arrived or can be added in the root before the image rows are passed. Cok [6] has suggested another approach of initially passing only the rows each transputer is responsible for and having the transputers exchange the bordering rows before beginning convolution. We have implemented pipeline versions of convolution using the first two schemes. The pipeline topology with eight computational transputers is shown in Figure 5.5.

Surprisingly, the more elegant first option proved to be no better than the second in spite of the fact that it requires less data to be passed. We believe that the extra overhead involved in passing the correct rows to each transputer in this case more than makes up for the time saved in the data passes. Timing results for various mask sizes for a 512×512 image for the second pipeline implementation are given in Figure 5.6. The times decrease dramatically as the number of processors increases from two to four to eight. The corresponding speedup graphs are given in Figure 5.7. The speedups are close to linear for all mask sizes. Notice also that the speedups increase as the mask size grows. Figure 5.8 shows the efficiency graphs for the second pipeline implementation for the mask sizes on the same 512×512 image. These efficiencies

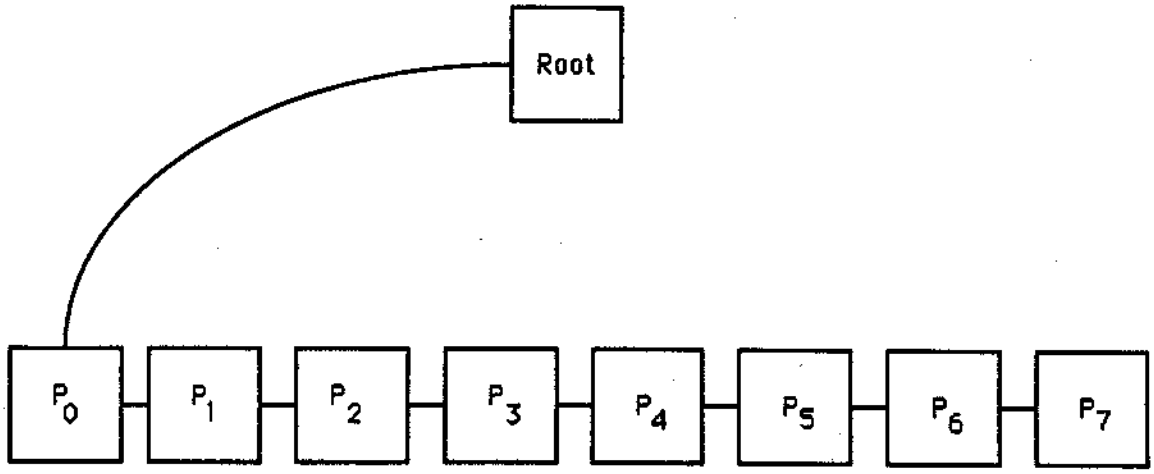

Figure 5.5: Eight Transputer Pipeline Topology

are all reasonably close to one.

5.4.3 Ring Implementation

In an attempt to reduce the time taken for data communication, we have created a ring version of the second pipeline implementation. The ring topology with eight computational transputers is shown in Figure 5.9.

The important feature of the ring implementation is that the root can distribute image rows and collect convolved rows in two directions at once. This cuts the maximum number of transputers on any one path to or from the root in half as compared to the pipeline version. Reductions in the time needed to distribute the image and to return the convolved image to the host are realized. Pseudo code for processor P_3 in the 8 transputer convolution ring program is given in Figure 5.10. *CONVOL* is defined in Section 3.5.1.

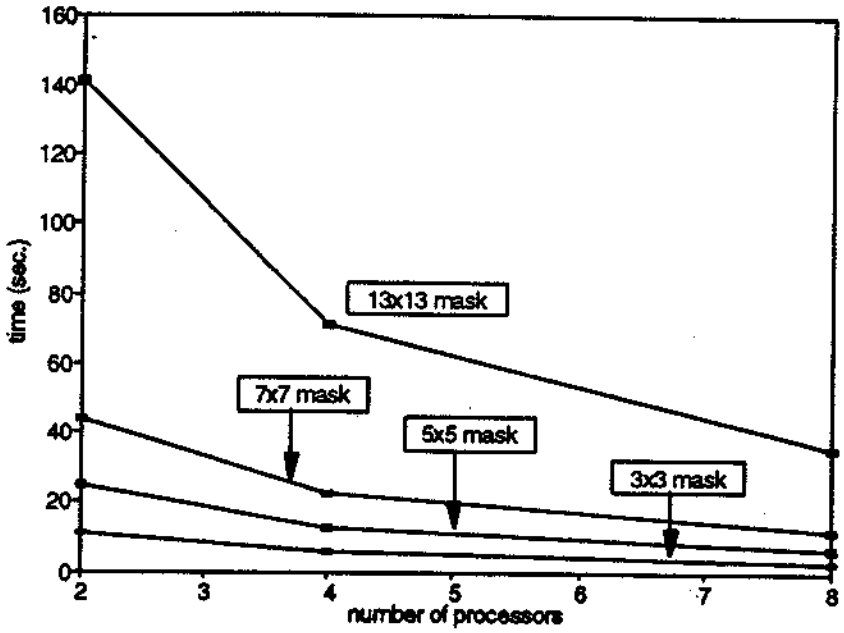

Figure 5.6: Convolution Pipeline: Timing Results

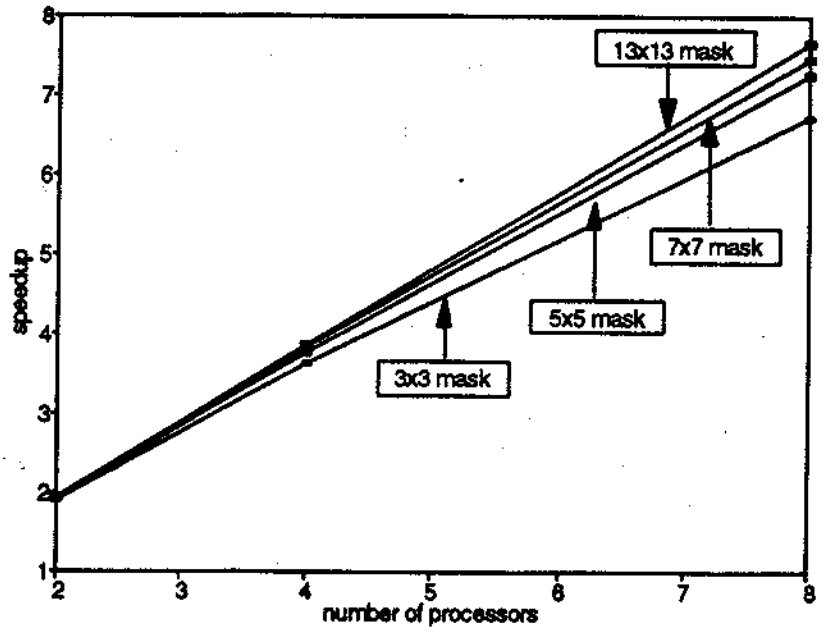

Figure 5.7: Convolution Pipeline: Speedup Graphs

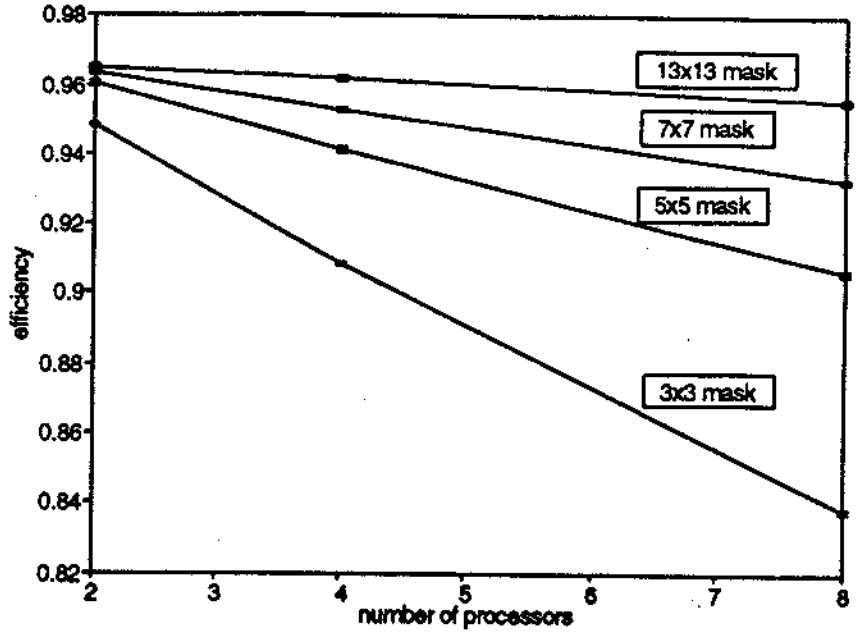

Figure 5.8: Convolution Pipeline: Efficiency Graphs

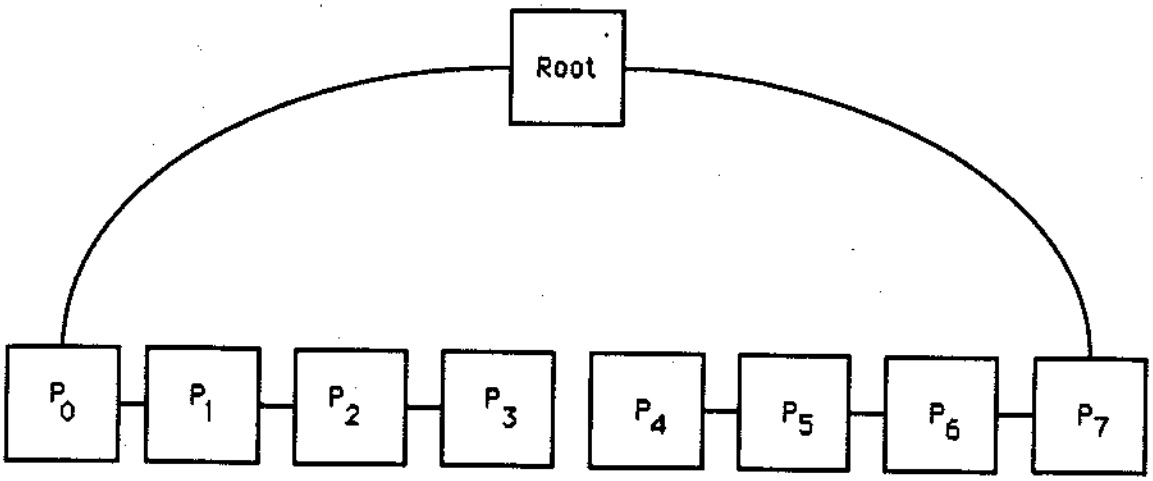

Figure 5.9: Eight Transputer Ring Topology

```
CONVOL8RINGP3
   get the image dimension N from P2
   get the mask dimension M from P2
   get the M × M mask G from P2
   get the zero padded image rows 3N/8 to N/2 + M − 2 F from P2
   CONVOL(F, N/8, N, G, M, C)
   send C to P2
END
```

Figure 5.10: Convolution Ring: Pseudo Code

The ring timing results for a 512×512 image are given in Figure 5.11. The results are quite similar to those of the second pipeline implementation. The ring implementation is actually only slightly faster than pipeline version. Speedups for these timing results are shown in Figure 5.12. The graphs are linear, as is the case with the pipeline version. However, the speedups are slightly greater, reflecting the improved timing results. Figure 5.13 contains the corresponding efficiency graphs. Again, the graphs are similar to the efficiency graphs for the pipeline version. However, the efficiencies are slightly greater than the pipeline efficiencies.

5.4.4 Tree Implementation

With the reduction of communication time again in mind, we next converted the ring implementation to an eight transputer tree implementation. The tree topology with eight computational transputers is shown in Figure 5.14. Processors P_0 and P_5 are the routers.

Although the maximum number of transputers on both paths from the root is the same as it is for the ring topology, the maximum path length from the root is shortened to three from four. However, in most cases the execution times for the eight transputer tree are no better than the corresponding times for the eight transputer ring. In fact, for a 13×13 convolution mask, the timing results are actually worse

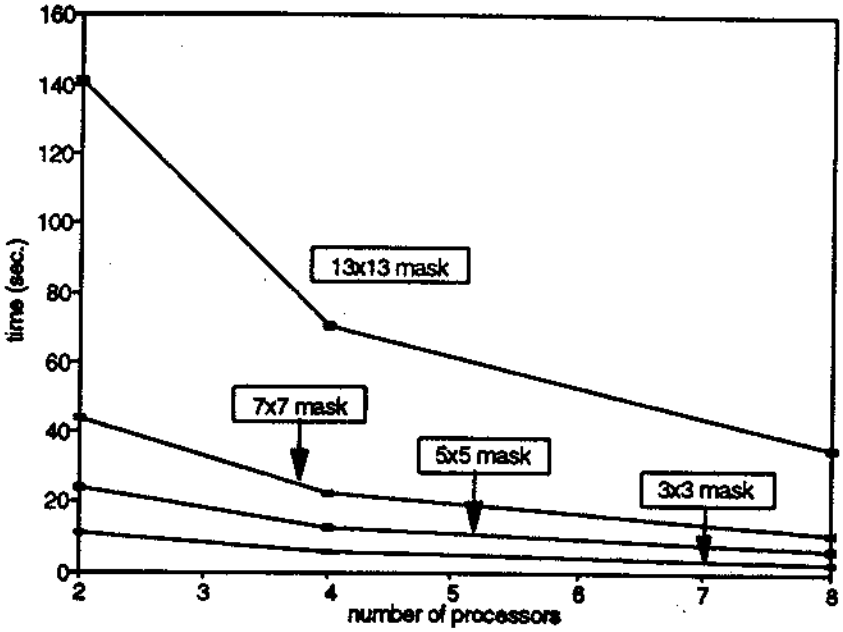

Figure 5.11: Convolution Ring: Timing Results

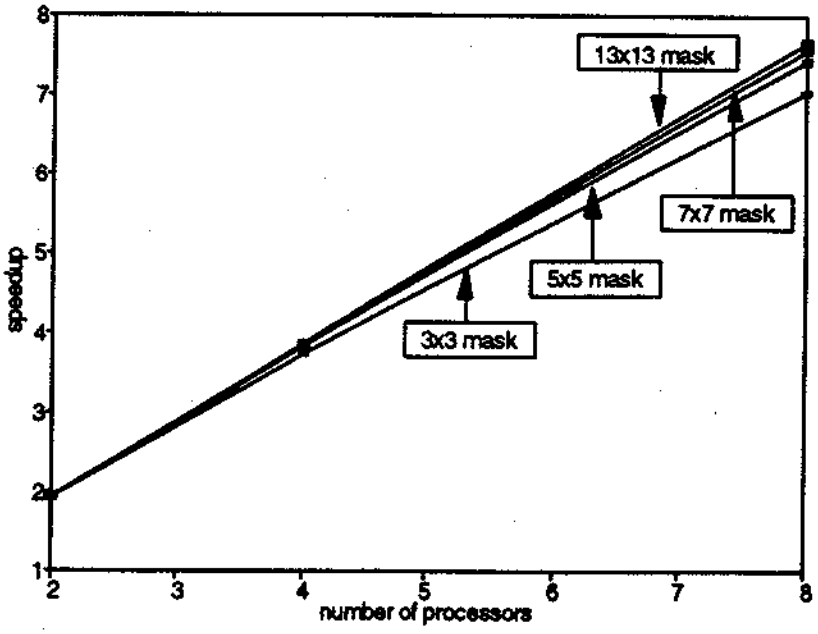

Figure 5.12: Convolution Ring: Speedup Graphs

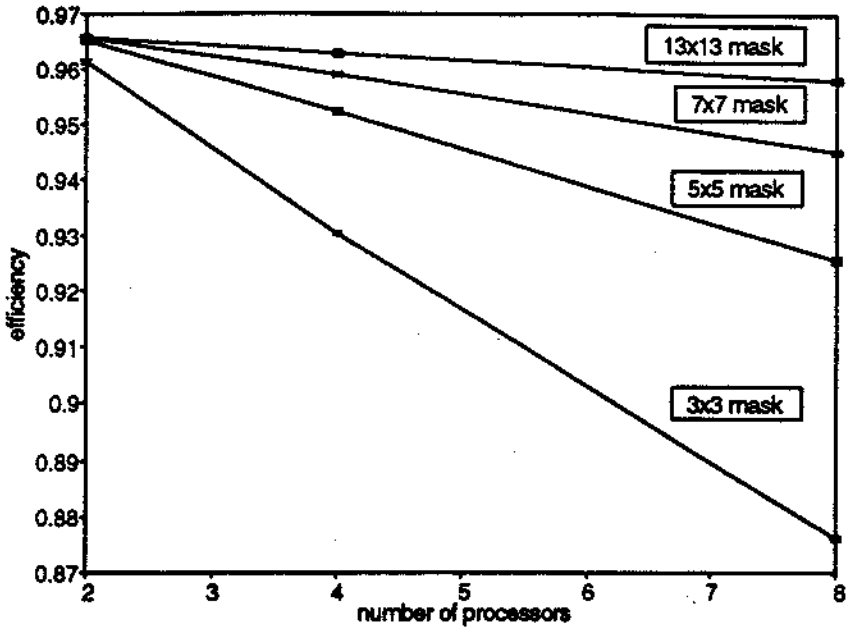

Figure 5.13: Convolution Ring: Efficiency Graphs

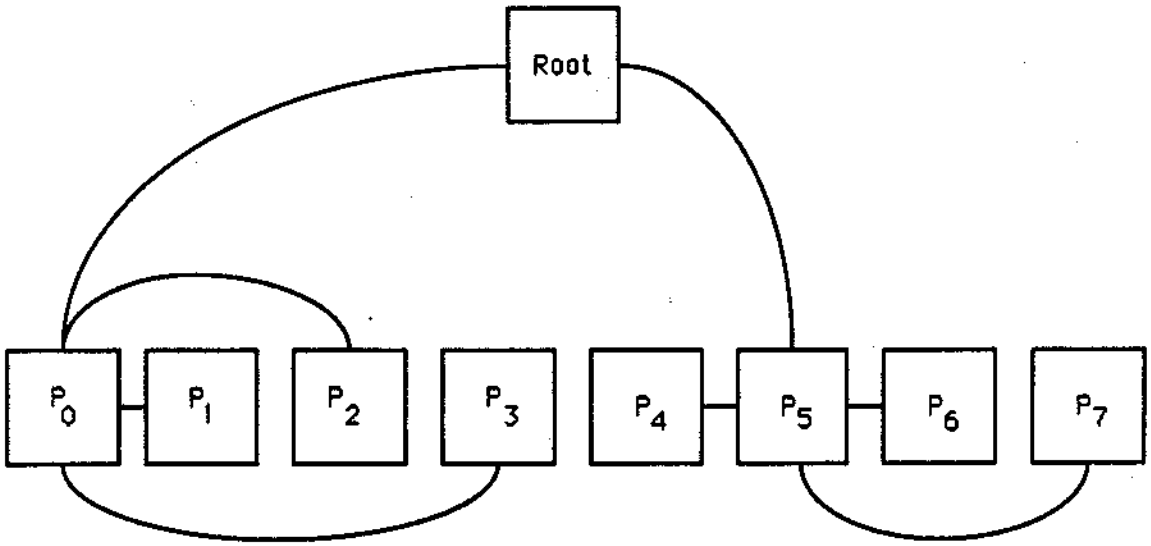

Figure 5.14: Eight Transputer Tree Topology

than the eight transputer *pipeline* results. Only when the program is run on a 3×3 mask on a 64×64 or a 128×128 image does the tree implementation run faster than the ring. Figure 5.15 illustrates another interesting feature of the convolution implementations. This figure shows efficiency graphs of the eight transputer tree implementation for various mask sizes. Clearly, efficiency increases as the image size increases.

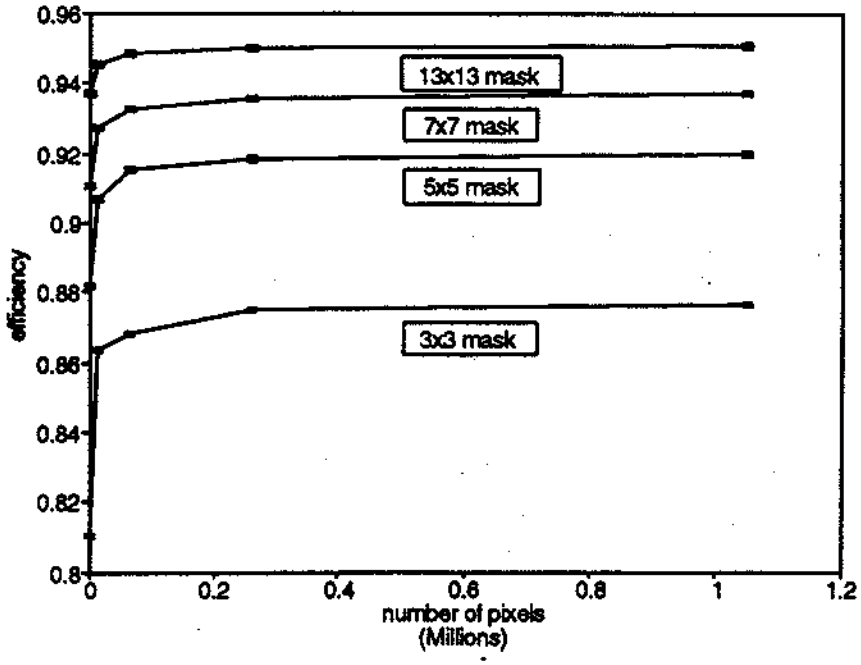

Figure 5.15: Convolution Tree: Efficiency Graphs

5.5 Regular Moments

The global nature of regular moment calculation made it a more challenging image processing technique to implement on transputers than convolution. It requires the full flexibility and reconnectivity of the transputer model. In this section we give descriptions and the timing results of our four multi-transputer implementations of

regular moments. As with convolution, we first give a description of the design of the implementations.

5.5.1 Design of the Implementations

In designing four regular moment implementations, we have again used the general strategies we have developed. The frame grabber transputer again passes packets of image rows to each computational transputer for processing. The tree topology is used for this initial data pass. The two dimensional recursive algorithm has the fastest single transputer implementation. We therefore considered using a data parallel version of the two dimensional recursive algorithm for moment calculation. Each transputer might calculate the sixteen two dimensional moments for its image rows using this algorithm. An instructional parallel algorithm could then be used to add those separate sets of moments to obtain the overall moments for the entire image. However, this technique does not work. The one dimensional recursive algorithm can certainly be used to calculate the row moments for each transputer's image rows. However, the one dimensional recursive algorithm cannot be used on these row moments to calculate the overall two dimensional moments of the rows.

Undaunted, we realized that we could still use the one dimensional recursive algorithm to calculate row moments for each transputer's rows. We also noted that the three algorithms with the fastest single transputer implementations (recursive, add and multiply, and partial sum) share another property besides speed. Each one separates the task of moment calculation into first calculating the moments of each image row and then using them to calculate the overall two dimensional image moments. We realized that we might exploit this in our implementations. We therefore have created three implementations which all use a data parallel version of the one dimensional recursive algorithm to calculate row moments on each computational transputer. These moments are then used by instructional parallel versions of the

three algorithms to calculate the overall image moments. We refer to each implementation according to the name of the instructional parallel algorithm used, namely recursive, add and multiply, and partial sum. The fourth multi-transputer regular moment implementation is an alternative version of the partial sum implementation. In all four implementations, the instructional parallel algorithm places the image moments in a transputer connected to the root. Additional transputer connections have been added to the tree structure so that the necessary data passing during the instructional parallel phase of the implementations can be accomplished. As is the case with the single transputer regular moment implementations, the moments are calculated as double precision numbers in the multi-transputer programs.

5.5.2 Recursive Implementation

The topology for the eight transputer version of the recursive implementation is given in Figure 5.16. The routers in the tree are processors P_0 and P_7. The extra connections made create a pipeline out of the computational transputers. Once the row moments have been calculated on each transputer, the one dimensional recursive algorithm is executed across the computational transputers as follows. (See Figure 3.8 for the pseudo code for the one dimensional recursive algorithm.) The rightmost computational transputer (P_7) begins by executing the one dimensional recursive algorithm on its row moments. Using the algorithm, P_7 determines not only the values of the sixteen overall moments $m_{KL}, K = 0..3, L = 0..3$ for its rows but also those of the middle eight overall moments $prevm_{KL}, K = 1..2, L = 0..3$ for the rows starting at its second row. When the algorithm ends, P_7 passes these 24 moments to the transputer on its left, P_6.

P_6 executes the one dimensional recursive algorithm on its own row moments with $M0 = m_{0L}, M1 = m_{1L}, M2 = m_{2L}, M3 = m_{3L}, PREVM1 = prevm_{1L}, PREVM2 = prevm_{2L}, L = 0..3$. When this is completed, the resulting 24 moments are passed on to P_5. This process continues down the pipeline of transputers. After each transputer

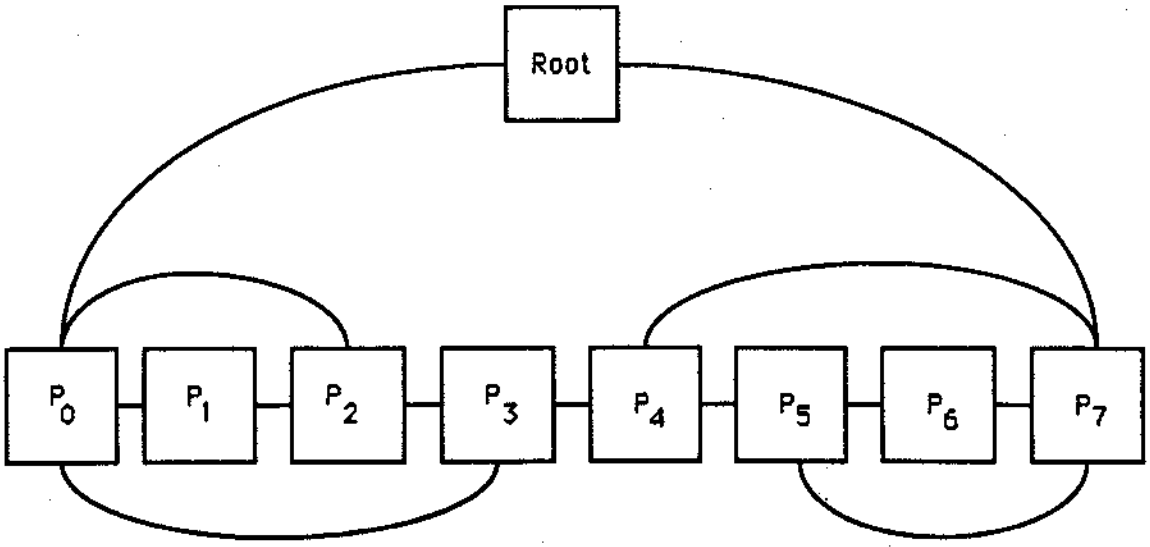

Figure 5.16: Eight Transputer Recursive Topology

has executed the algorithm, the overall moments for the part of the image starting at its first row have been calculated. When the algorithm has executed on the leftmost transputer P_0, the sixteen overall image moments have been calculated. At this time, P_0 passes the image moments back to the root for output.

Figure 5.17 shows the timing results for various image sizes for the recursive implementation. As might be expected, the execution times do decrease as the number of transputers used increases. The speedup results corresponding to these execution times are shown in Figure 5.18. In contrast with the convolution implementations, these speedups are clearly less than linear. Figure 5.19 shows the efficiency graphs of the recursive implementation for the same image sizes. The efficiency of the implementation clearly increases as the image size grows. The rapid decline of the efficiency of the implementation as the number of processors used increases is also evident. This decline is due to the fact that increasing the number of transputers does reduce the

time needed to calculate the row moments but actually increases the time taken to calculate the overall moments. This is because during the overall moment calculation only one computational transputer is active at a time and adding transputers only serves to increase the number of moment data passes required in this last instructional parallel stage of the computation. This is the biggest drawback of the recursive implementation. Perhaps if all the computational transputers work together and at once on calculating the overall moments a faster implementation would result. The partial sum and add and multiply implementations attempt to make use of this idea.

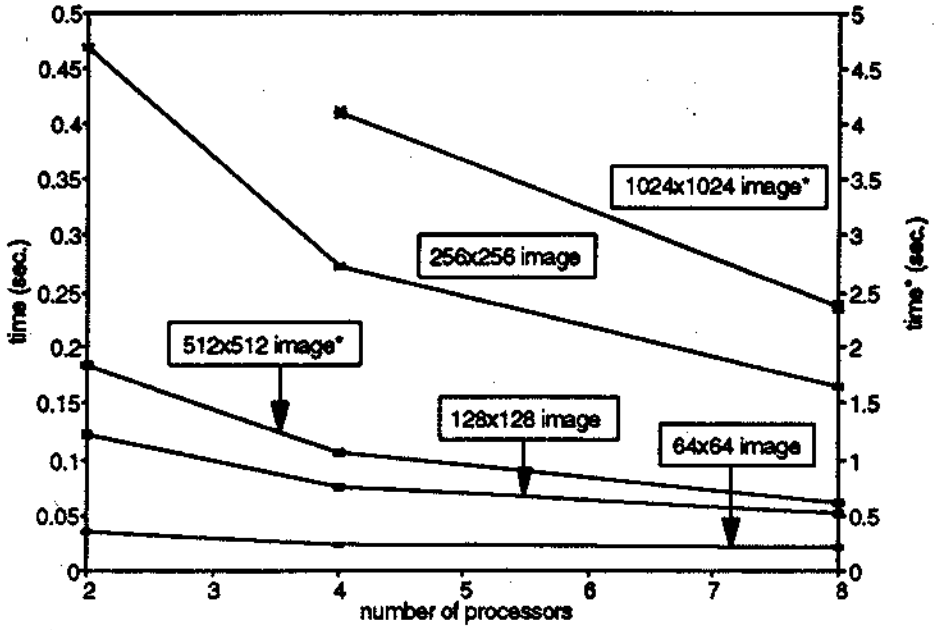

Figure 5.17: Recursive Timing Results

5.5.3 Partial Sum Implementations

It was necessary to augment the basic tree structure heavily for the eight transputer partial sum implementation. P_0 and P_7 are the router transputers in the tree. The following connections between computational transputers were made unless they were

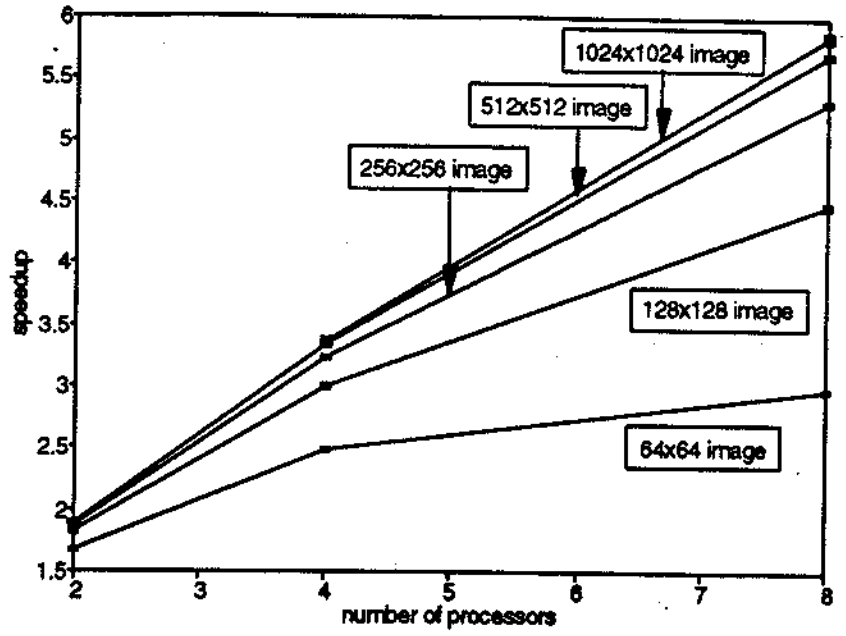

Figure 5.18: Recursive Speedup Graphs

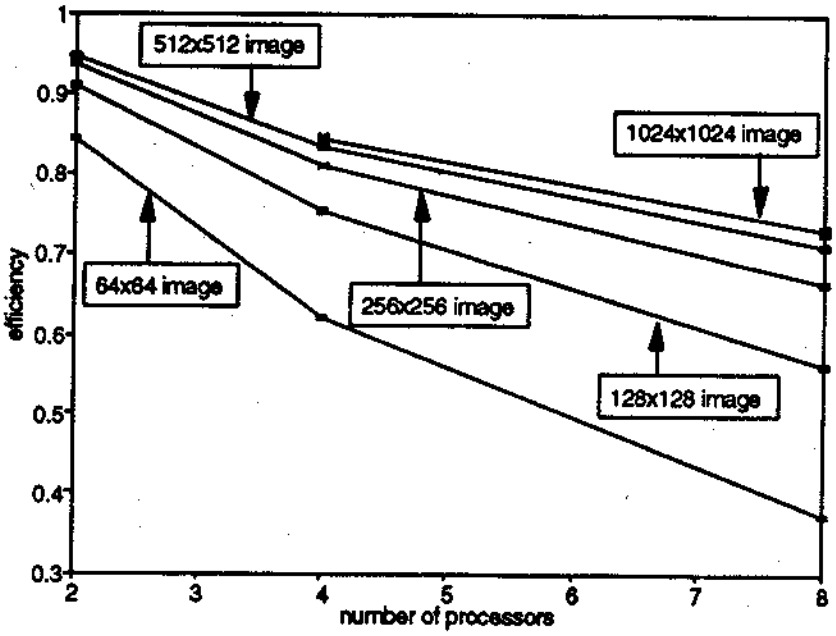

Figure 5.19: Recursive Efficiency Graphs

part of the basic tree structure. First of all, the transputers were joined into a pipeline. Secondly, with the exception of the third (P_2) and fourth (P_3) transputers, each transputer was connected to one two before or after it. Lastly, every transputer was joined to the transputer four positions before or after it. The resulting topology (shown in Figure 5.20) seems formidable indeed.

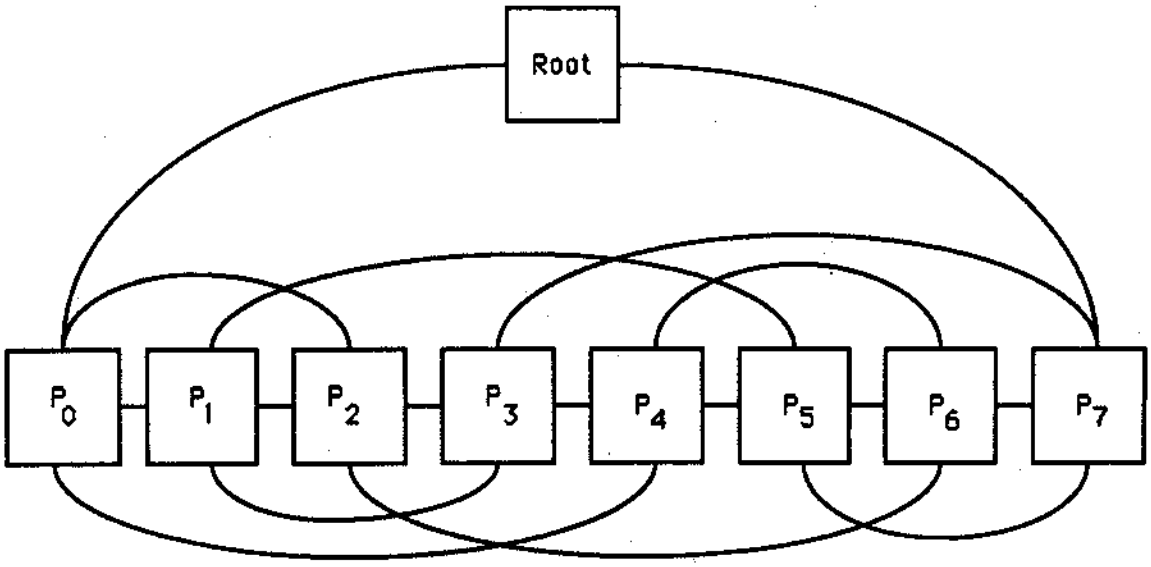

Figure 5.20: Eight Transputer Partial Sum Topology

In the instructional parallel part of the partial sum implementation the one dimensional partial sum algorithm executes across the computational transputers. As we explained in Section 3.6.2, a series of pairwise additions is performed in the inner loop of the algorithm. First, each pixel is added to the preceding pixel. Then it is added to the pixel two pixels before it. Next it is added to the pixel four before it, and so on. After each addition, the distance between added pixels doubles until the distance is half the width of the image.

This is basically what occurs when the instructional phase of the partial sum implementation executes. In this case, however, row moments are added together. The row moments are spread out over many transputers. So the pairwise additions require the transfer of row moments between transputers. In the eight transputer implementation, the first pairwise additions require each computational transputer (except the first, leftmost transputer P_0) to pass one moment to the processor preceding it. That is why the pipeline topology is necessary. The next pairwise additions require two moments to be sent, the additions after that four and so on. However, when the width between the moments to be added exceeds the number of rows r processed on each transputer, it becomes necessary for transputers separated by two to exchange moments. The fourth transputer (P_3) must act as a router for moment transfers between the third (P_2) and fifth (P_4) transputers because of lack of links. Similarly, P_4 acts as a router between P_3 and P_5. For the last pairwise additions, transputers receive r moments from the transputer four positions after them. The last two data exchanges explain the need for the last connections described in the first paragraph of this section.

The above series of pairwise additions is executed three times on the row moments. In this way, the first three overall moments corresponding to the row moments are calculated. The fourth overall moment is then calculated. To do so, the row moments are first added pairwise. Finally, the row moments are added up in a similar, but simpler way than the above full series of pairwise additions. In order to calculate the first sixteen overall image moments, the entire above process is repeated for all four sets of row moments. The first transputer (P_0) collects the overall moments as they are calculated.

We have also implemented an alternative version of this partial sum implementation. In this implementation, the partial sum process is done for all four sets of row moments at once. This reduces the number of data transfers (if not the amount of

data sent) by a factor of four. In spite of this reduction, this more elegant implementation is no faster than the simpler method. We attribute this to the larger overhead in buffer management that it requires compared to the first method.

Figure 5.21 contains the timing results for the first partial sum implementation. As with the recursive implementation, increasing the number of transputers results in decreases in execution times for all image sizes. The times are all actually slightly slower than those of the recursive implementation, however. The partial sum speedup graphs are similar to those of the recursive program. They are given in Figure 5.22. These graphs are again clearly less than linear. The corresponding efficiency graphs are given in Figure 5.23. The reader should notice that this implementation is also most efficient for larger images. Once more we see a quick decline in efficiency as the number of transputers used grows. The reason for the decline is less obvious than it was in the recursive case. Increasing the number of transputers again decreases the amount of time needed to calculate the row moments. It also decreases the time needed to perform the additions required in the partial sum stage of the computation which calculates the overall moments. However, the partial sum technique requires a large amount of data communication when executed on more than one transputer. As the number of transputers increases, this requirement increases dramatically. To make matters worse, for the eight transputer program, routers (P_3, P_4) are necessary because of lack of links. This results in a communication bottleneck which further slows down the last stage of the computation. The observed sublinear speedups and decreasing efficiencies make sense in light of this information.

5.5.4 Add and Multiply Implementation

In the description of our design of the regular moment implementations, our disappointment at the impossibility of a pure data parallel version of the two dimensional recursive algorithm to calculate overall moments was expressed. We could not simply

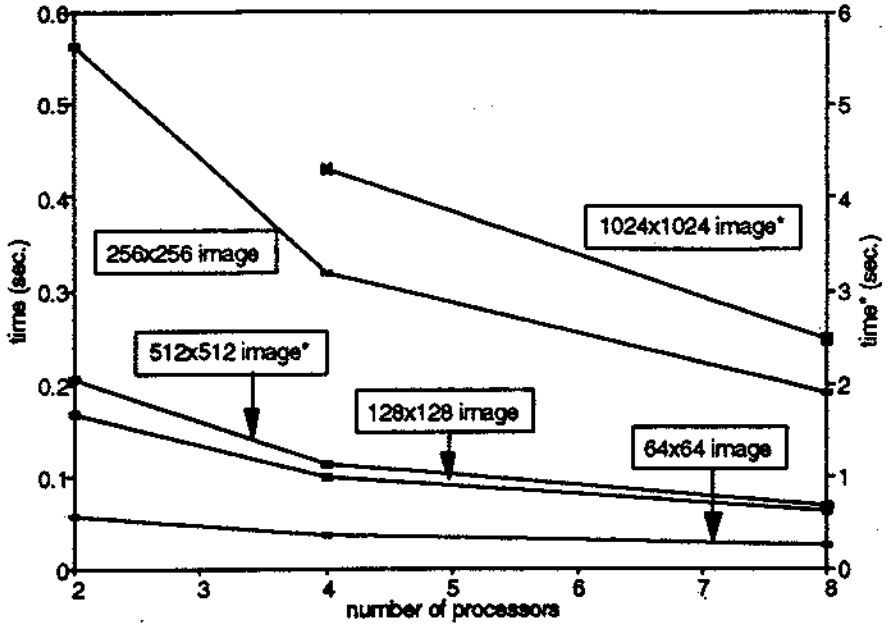

Figure 5.21: Partial Sum Timing Results

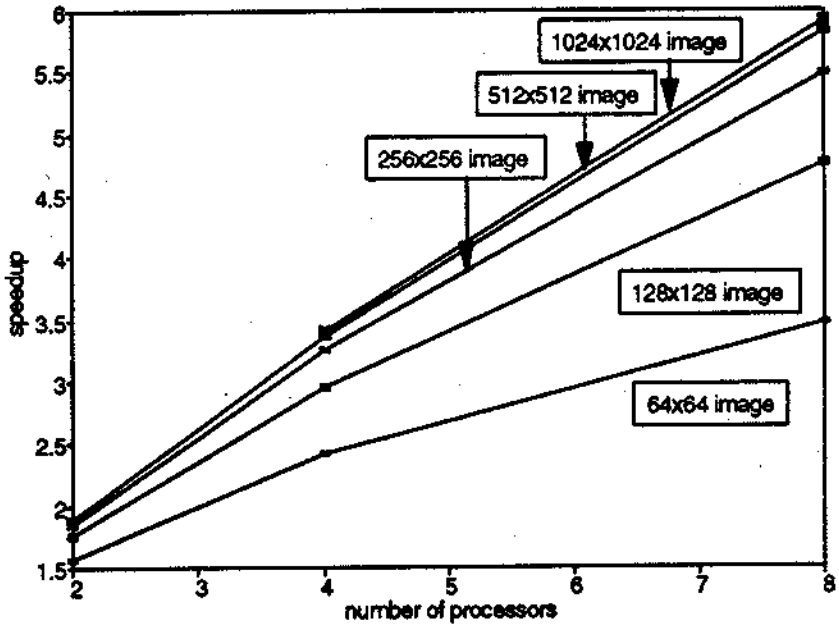

Figure 5.22: Partial Sum Speedup Graphs

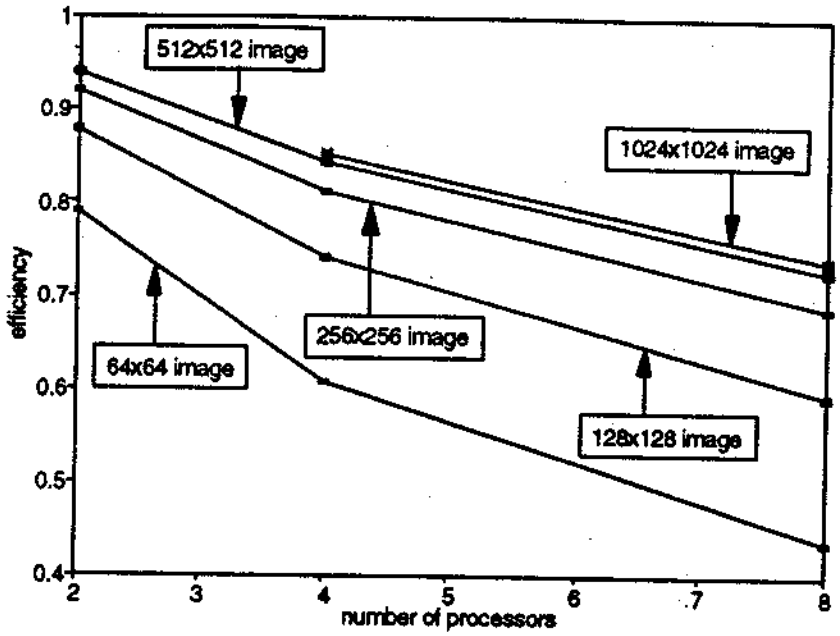

Figure 5.23: Partial Sum Efficiency Graphs

use it to calculate the overall moments for each transputer's rows and add the resulting sets of moments together to get the moments for the entire image. The add and multiply algorithm, however, can be used in this way to arrive at the overall image moments. The topology required is relatively simple compared to that needed for the partial sum program. The eight transputer version is shown in Figure 5.24. It is simply a tree structure with the fifth (P_4) and seventh (P_6) computational transputers also connected. P_0 and P_7 are again the router transputers.

Once the row moments have been calculated in the add and multiply implementation, a data parallel version of the one dimensional add and multiply algorithm is executed simultaneously on the row moments on each computational transputer. The offset of the first row each transputer processes is used. When this has been done, the last technique used to add each set of row moments in the partial sum implementations is used to add the sets of 16 moments. It is for this addition that the extra

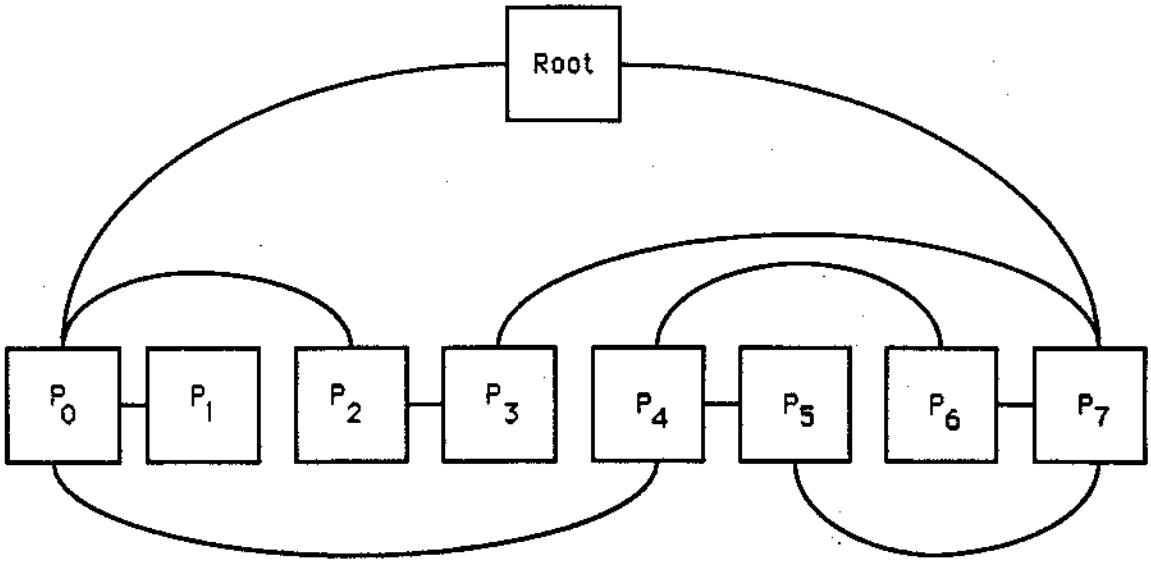

Figure 5.24: Eight Transputer Add and Multiply Topology

connection in the topology is needed. The first transputer (P_0) ends up with the moments and passes them back to the root for host output. In Figure 5.25 pseudo code is given for processor P_i in the 8 transputer add and multiply implementation. Both *RECURSE* and *ADDANDMU* are defined in Section 3.6.2.

In Figure 5.26 the add and multiply timing results for various image sizes can be found. Again, as the number of processors increases, the times all decrease. Figure 5.27 shows the speedup graphs for these times. The reader can verify that these graphs again flatten out as the number of processors increases. Efficiency graphs for the image sizes are shown in Figure 5.28. The efficiency graphs also drop as the number of transputers increases.

```
ADDANDMU8P4
   get the image dimension N from P0
   get the image rows N/2 to 5N/8 − 1 F from P0
   1DRECURSE(F, N/8, N, RM)
   1DADDANDMU(RM, 4, N/8, N/2, m)
   get 16 overall moments P5mom from P5
   for k = 0..3
       for l = 0..3
           m_kl+ = P5mom_kl
       end for
   end for
   get 16 overall moments P6mom from P6
   for k = 0..3
       for l = 0..3
           m_kl+ = P6mom_kl
       end for
   end for
   send 16 overall moments m to P0
END
```

Figure 5.25: Add and Multiply Pseudo Code

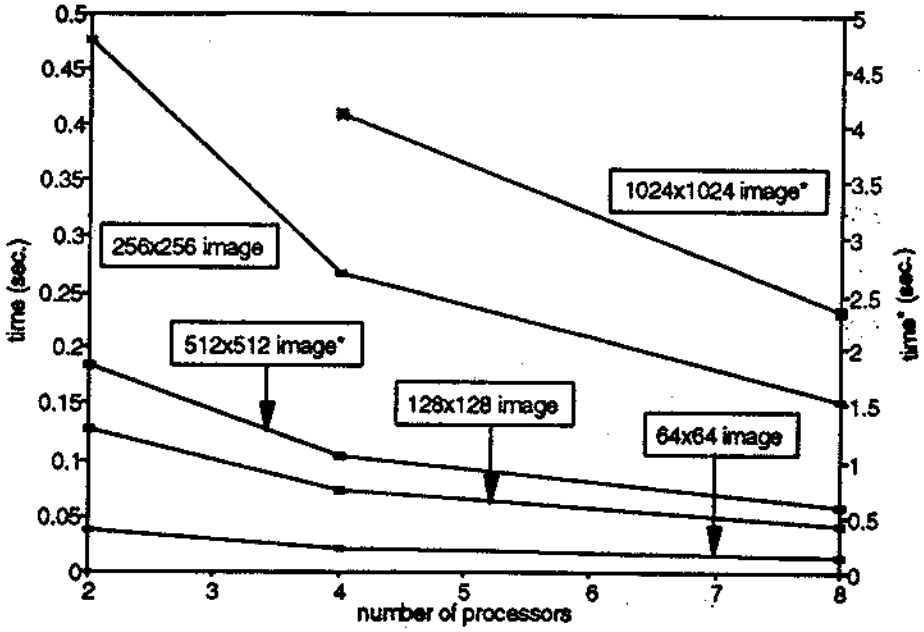

Figure 5.26: Add and Multiply Timing Results

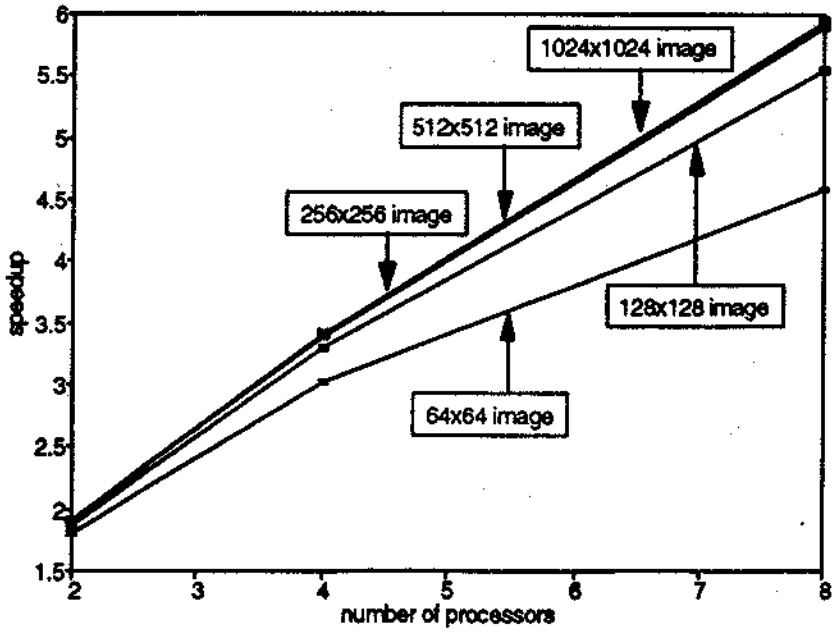

Figure 5.27: Add and Multiply Speedup Graphs

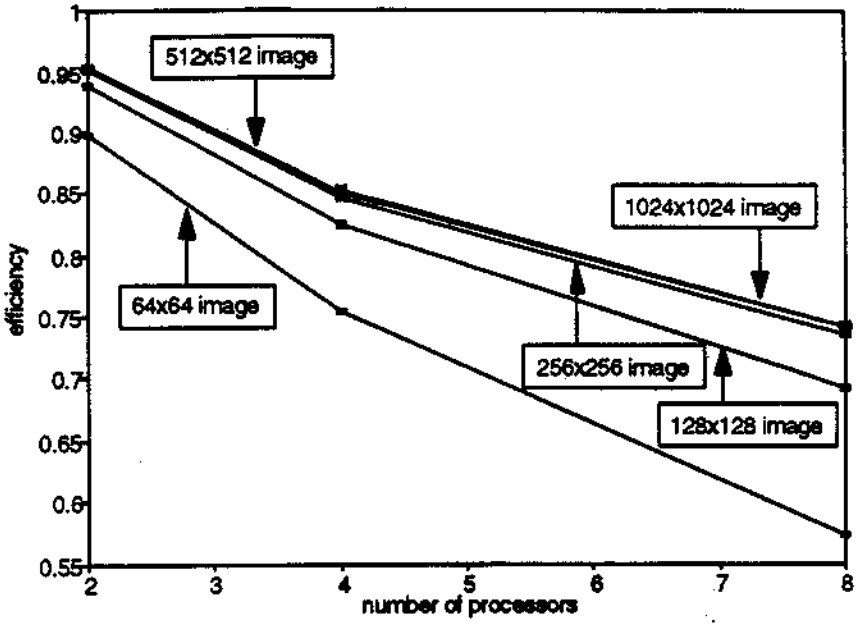

Figure 5.28: Add and Multiply Efficiency Graphs

5.6 Analysis of the Implementations

5.6.1 Convolution

It is most important to note that the convolution implementations all have close to linear speedups. This is typical of local techniques with pure data parallel implementations which require no communication between processors during the computational stage. Some observations about topology selection are also indicated. The transputer topology is varied over the last three convolution implementations. The time needed to pass the original and convolved images is partly determined by the topology used. Yet the timing results for these implementations are quite similar. We conclude from this that for pure data parallel image processing algorithms implemented on the B008, topology selection is relatively unimportant. However, for another transputer board with more slots for transputers, topology selection may be more important, since the

paths along which the image segments are passed may be longer.

Lastly, we present time complexity analysis of the computational stage of the implementations. During the computational stage, $\frac{N^2M^2}{P}$ floating point multiplications and additions are executed on each of P processors. For an N processor array where each processor is responsible for performing convolution on one row of the image, the parallel asymptotic complexity of our convolution implementations will be $O(NM^2)$ floating point multiplications and additions.

5.6.2 Moments

In sharp contrast with the convolution implementations, our regular moment implementations suffer from flattening speedups and decreasing efficiencies as the number of transputers increases. The global nature of moments entails that as the number of processors used to calculate them increases, so too does the amount of communication needed among the processors. It is this communication increase that has resulted in the decreased efficiencies of our four and eight transputer moment implementations. Additionally, although we did not vary topologies for our moment implementations, we feel that the moment programs would be quite sensitive to changes in topology. This is also clearly different from the convolution implementations.

In 1990, Chen [7] proposed efficient parallel regular moment algorithms for both linear and two dimensional processor arrays. The algorithm proposed for the linear array is the multi-transputer partial sum technique we have implemented. For a two dimensional array of processors, each of which is given a section of the image, Chen recommends that the processors first collectively use the partial sum algorithm to calculate the row moments and again use the partial sum procedure on these moments to calculate the overall image moments. In a more recent article, Pan [17] showed that because Chen forgot to take the time needed for data passing into account, the theoretical timing analysis presented in his article is incorrect. In so doing, Pan demonstrated that Chen's two dimensional algorithm implemented on an

$N \times N$ array of processors would have the same parallel asymptotic time complexity of $O(N)$ as his one dimensional algorithm implemented on an N processor linear array when calculating the regular moments of an $N \times N$ image.

The recursive implementation presented in this chapter requires $\frac{8N^2}{P}$ double precision additions to calculate the row moments of an $N \times N$ image on an array of P processors using the one dimensional recursive algorithm. It requires an additional $32N$ double precision additions and $24P$ double precision data transfers to calculate the first 16 overall moments. In particular, for an array of N processors, the algorithm requires $40N$ additions and $24N$ data transfers. Therefore, when implemented on such an array, the algorithm clearly requires $O(N)$ double precision additions and data transfers.

The add and multiply algorithm, on the other hand, requires the same $\frac{8N^2}{P}$ double precision additions to calculate the row moments of an $N \times N$ image on an array of P processors. In addition, $\frac{16N}{P}$ double precision multiplications and additions are needed to calculate the first 16 overall moments of the image rows of each processor. To add these up, it takes $\sum_{i=0}^{\log_2^P - 1} 2^i 16 = 16(P-1)$ double precision data transfers and $16\log_2^P$ additions. For an array of N processors, the add and multiply algorithm clearly will also have linear asymptotic complexity in the number of double precision additions and data transfers.

We believe that the behaviour of an implementation is most significant as image size grows and the number of transputers used increases. For the largest image size of 1024×1024, the execution time of the eight transputer add and multiply program is 2.351 seconds and that of the recursive implementation 2.374 seconds. These times excel the corresponding time of 2.484 seconds for the partial sum implementation.

This indicates that when implemented on the B008 in Parallel C, the add and multiply and the recursive multi-transputer algorithms are superior to the partial sum algorithm proposed by Chen. We have shown that the algorithms have the same linear parallel time complexity as the partial sum algorithm for a linear array

of N SIMD or MIMD processors. In his article, Pan correctly points out the wide range of technologies implementing the SIMD and MIMD architectures. Which of the three algorithms is best for any given technology can only be determined by experimentation.

A last observation is that the first convolution pipeline implementation and the alternative partial sum implementation indicate that reducing the amount of data transferred in an implementation or the number of transfers required does not always result in faster multi-transputer programs. We believe that the Parallel C software overhead needed to make these reductions may cancel out their benefits.

5.7 Remarks

The single transputer convolution program was changed so that an integer mask was processed rather than a floating point mask. It is also possible to change the multi-transputer convolution implementations so that integer masks are used. We feel that faster execution times would again result if this were done. Similarly, the multi-transputer programs can be changed to produce floating point moments rather than double precision ones. The execution times for the altered implementations would of course be less than their double precision analogues. Also, as in the single transputer case, we have restricted timing results and analysis given to $N \times N$ images with N equivalent to powers of two. To maintain an even computational load, the multi-transputer programs were run with two, four or eight computational processors. The convolution implementations required little modification to change from two to four to eight transputers. The moment programs however, had separate implementations for each different number of transputers.

Chapter 6

Performance Modelling

6.1 Introduction

In this chapter, we present performance modelling of two of our multi-transputer implementations. The modelling is intended to confirm that the timing results for the implementations correspond to what their design indicates should be expected. We believe the reasoning process presented in Chapter 5 that led to the designs is sound and stands well on its own. We are not trying to prove its legitimacy. Instead, we are trying to demonstrate that we in fact understand what is occurring during the implementations. Additionally, we want to create a scientific method with which we can predict the behaviour of our implementations for a greater number of transputers. The modelling is high-level, coarse modelling rather than an exact bit-by-bit analysis. In particular, we ignore the overhead incurred with Parallel C inter process communication and process switching time on each transputer. Since all the implementations have the same basic structure (*i.e.*, an initial data pass followed by a computational stage and, in the case of convolution, a final data pass) we have chosen to model only the best implementation for convolution (ring) and regular moments (add and multiply). The models for the other implementations would be similar.

6.2 Convolution Model

6.2.1 Notation

In our analysis, the following notation is used:

P denotes the number of transputers used.

N denotes the image size.

M denotes the mask size.

$T_{tl}(N, M, P)$ denotes the total time in seconds needed to convolve an $N \times N$ image with an $M \times M$ mask using the ring implementation with P computational transputers.

$T_{cp}(N, M, P)$ denotes the computation time in seconds needed to convolve an $N \times N$ image with an $M \times M$ mask using the ring implementation with P computational transputers.

$T_{cm}(N, M, P)$ denotes the data communication time in seconds needed to convolve an $N \times N$ image with an $M \times M$ mask using the ring implementation with P computational transputers.

$T_{sn}(N, M)$ denotes the time in seconds needed to convolve an $N \times N$ image with an $M \times M$ mask using the single transputer convolution program.

$T_{ps}(B, T)$ denotes the time in seconds needed to send B bytes to each of T transputers on a path from the root using the packet method.

6.2.2 Computation Time

In the single transputer convolution program, N^2M^2 floating point additions and N^2M^2 multiplications and $2N^2M^2$ array references are executed. In addition, there

are $2N^2M$ array offset additions performed. Finally, N^2 function calls and array references are needed. We ignore operations that occur less than N^2 times in our analysis. Therefore,

$$T_{sn}(N, M) \simeq k_1 N^2 M^2 + k_2 N^2 M + k_3 N^2$$

The value of k_1 seems to be approximately 5.91×10^{-6}, that of k_2 to be 3.26×10^{-6}, and k_3 to be 1.26×10^{-5}. k_3 was determined by timing the program with only the N^2 loop executing. This was accomplished by running the program with the NM^2 loop commented out. k_3 was set to be the execution time divided by N^2. Next, k_2 was set to be the execution time of the NM^2 loop divided by NM^2. To get this time, the execution time of the N^2 loop was subtracted from the execution time of the NM^2 loop and the N^2 loop. Lastly, k_1 was set to be the execution time of the N^2M^2 loop divided by N^2M^2. Substituting these constant values, we get

$$T_{sn}(N, M) \simeq 5.91 \times 10^{-6} N^2 M^2 + 3.26 \times 10^{-6} N^2 M + 1.26 \times 10^{-5} N^2$$

These computations are spread evenly among the P computational transputers used. Therefore, we should get

$$T_{cp}(N, M, P) = \frac{T_{sn}(N, M)}{P}$$

$$= \frac{5.91 \times 10^{-6} N^2 M^2 + 3.26 \times 10^{-6} N^2 M + 1.26 \times 10^{-5} N^2}{P}$$

6.2.3 Communication Time

Each image consists of N^2 pixels. In our multi-transputer implementations, each computational transputer is passed $1/P$ of the pixels or $\frac{N^2}{P}$ pixels. (This actually is a lower bound for the convolution ring, since bordering pixels are also passed to each transputer.) In the ring implementation, the root passes the pixels to each of $P/2$ transputers down both branches of the ring at the same time using the packet

method. This takes $T_{ps}(\frac{N^2}{P}, \frac{P}{2})$. The root must first separate the image into P packets for passing. This takes time proportional to the number of pixels in the array, N^2. Additional time is required to send the packets down the line of transputers. This takes time proportional to the number of bytes in the packet ($\frac{N^2}{P}$) and to the number of transputers in the line ($P/2$). Therefore,

$$T_{ps}(\frac{N^2}{P}, \frac{P}{2}) \simeq k_1 N^2 + k_2(\frac{N^2}{P})(\frac{P}{2})$$

By experimentation with the data passing program that generated the packet timing results given in Figure 5.2, we have determined that k_1 seems to be about 2.25×10^{-7}, k_2 about 6.42×10^{-7}. Substituting, we get,

$$T_{ps}(\frac{N^2}{P}, \frac{P}{2}) \simeq 2.25 \times 10^{-7} N^2 + 6.42 \times 10^{-7}(\frac{N^2}{P})(\frac{P}{2})$$

$$= 2.25 \times 10^{-7} N^2 + 3.21 \times 10^{-7} N^2$$

$$= 5.46 \times 10^{-7} N^2$$

The data pass is repeated in reverse at the end of the program when the convolved image is passed back to the root. Therefore, the total communication time,

$$T_{cm}(N, M, P) \simeq 2T_{ps}(\frac{N^2}{P}, \frac{P}{2})$$

$$= 2 \times 5.46 \times 10^{-7} N^2$$

$$= 1.09 \times 10^{-6} N^2$$

6.2.4 Overall Time

We have

$$T_{tl}(N, M, P) \simeq T_{cp}(N, M, P) + T_{cm}(N, M, P)$$

Therefore,

$T_{tl}(N, M, P) \simeq \frac{5.91\times10^{-6}N^2M^2+3.26\times10^{-6}N^2M+1.26\times10^{-5}N^2}{P} + 1.09 \times 10^{-6}N^2$.

In Figure 6.1, the last function is plotted for $P = 2, 4, 8$ with $N = 512$ and $M = 7$. The actual timing results for the ring implementation are also plotted. The actual times are slightly more than the estimated times.

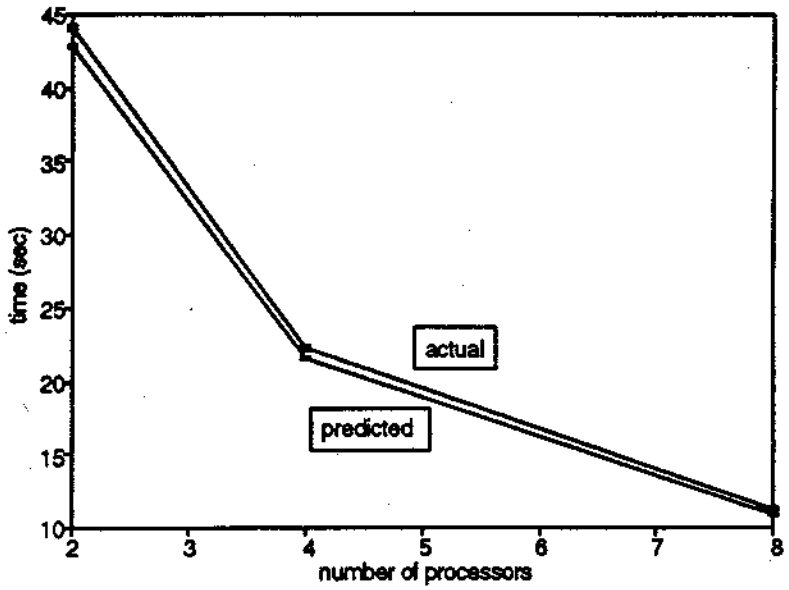

Figure 6.1: Convolution Model Graphs

6.3 Regular Moment Model

6.3.1 Notation

The notation used in this model is similar to that used for the convolution ring model. In particular,

P denotes the number of transputers used.

N denotes the image size.

$T_{tl}(N,P)$ denotes the total time in seconds needed to calculate the first 16 moments of an $N \times N$ image using the add and multiply implementation with P computational transputers.

$T_{cp}(N,P)$ denotes the computation time in seconds needed to calculate the first 16 moments of an $N \times N$ image using the add and multiply implementation with P computational transputers.

$T_{cm}(N,P)$ denotes the data communication time in seconds needed to calculate the first 16 moments of an $N \times N$ image using the add and multiply implementation with P computational transputers.

$T_{sn}(N)$ denotes the time in seconds needed to calculate the first 16 moments of an $N \times N$ image using the single transputer hybrid recursive/add and multiply program.

$T_{ps}(B,T)$ denotes the time in seconds needed to send B bytes to each of T transputers on a path from the root using the packet method.

6.3.2 Computation Time

In the single transputer implementation of the recursive/add and multiply hybrid algorithm, $8N^2$ additions and N^2 array references are required. There are also $16N$

multiplications and additions and N array offset additions needed. So we get

$$T_{sm}(N) \simeq k_1 N^2 + k_2 N$$

The empirical values are $k_1 \simeq 1.32 \times 10^{-5}$ and $k_2 \simeq 1.66 \times 10^{-4}$. These were determined in a similar manner to the computational constants in the convolution model. The program was timed with only the N loop executing, and the result was divided by N to get k_2. To get the constant k_1, the execution time of the N^2 loop was timed and divided by N^2. Substituting the constants, we get

$$T_{sm}(N) \simeq 1.32 \times 10^{-5} N^2 + 1.66 \times 10^{-4} N$$

In the last stage of the add and multiply implementation, the 16 moments calculated by the P processors are added up in partial sum fashion across the network. This should take time proportional to $\log_2 P$, *i.e.*, this time equals $k_1 \log_2 P$ for some k_1. For a 512×512 image, k_1 seems to be 1.47×10^{-2}. As with the convolution ring implementation, all computation except this last addition is spread evenly among the P computational transputers. So,

$$T_{cp}(N,P) \simeq \frac{T_{sm}(N)}{P} + 1.47 \times 10^{-2} \log_2 P$$

$$= \frac{1.32 \times 10^{-5} N^2 + 1.66 \times 10^{-4} N}{P} + 1.47 \times 10^{-2} \log_2 P$$

6.3.3 Communication Time

In the add and multiply implementation, the image is distributed to the P transputers using the packet method as is the case with the ring convolution implementation. (No bordering pixels are passed in this case, however.) This again takes $T_{ps}(\frac{N^2}{P}, \frac{P}{2})$. As in the convolution ring, this should be approximately $5.46 \times 10^{-7} N^2$ seconds. In the regular moment implementations, however, there is no convolved image to pass back to the root as is the case with convolution. Therefore, the total communication time,

$$T_{cm}(N,P) \simeq 5.46 \times 10^{-7} N^2$$

6.3.4 Overall Time

We know that $T_u(N,P) \simeq T_{cp}(N,P) + T_{cm}(N,P)$. So,

$$T_u(N,P) \simeq \frac{1.32 \times 10^{-5} N^2 + 1.66 \times 10^{-4} N}{P} + 1.47 \times 10^{-2} \log_2 P + 5.46 \times 10^{-7} N^2$$

In Figure 6.2, the above function is plotted for $P = 2, 4, 8$ with $N = 512$. The actual timing results for the add and multiply implementation are also plotted. This time our estimates are slightly greater than the actual values.

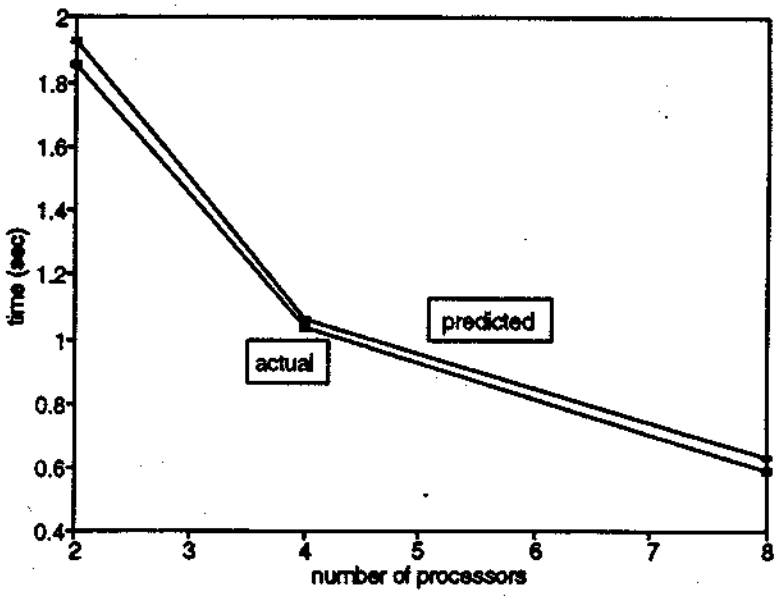

Figure 6.2: Regular Moment Model Graphs

6.4 Remarks

Since the parameters were chosen for runs where $N = 512$ and $M = 7$, the models work best for these values of N and M. We believe, however, that the models may also work well for other values of N and M. Since the data passing programs

do not allow for the simultaneous communication and computation possible in the transputer model, the data passing parameters have been set slightly higher than the data passing timing results might indicate they should be. Because of this, our data passing model may differ slightly from what occurs in a normal computational multi-transputer program. While not perfect, we believe that the models are close enough to the observed timing results (within 5%) to confirm that our understanding of the behaviour of the two programs is correct.

Chapter 7

Conclusion

7.1 Thesis Summary

In this thesis, we have presented parallel implementations of the image processing techniques of convolution and regular moments on multiple transputers programmed with Parallel C. Several single transputer programs were first created in preparation for multi-transputer programming. Eight multi-transputer programs have been presented and analyzed. In all, we have created 21 different transputer programs consisting of over 100 source files and approximately 20,000 lines of code.

Our results are as follows. Most importantly, we observed linear speedups for all the multi-transputer convolution programs. The multi-transputer regular moment programs are also considerably faster than their single transputer counterparts. However, as the number of transputers increases, the efficiencies of the programs decrease at a greater rate than in the convolution implementations. This is a symptom of the increase in communication overhead that parallel implementations of global techniques undergo as the number of processors increases. Local techniques, such as convolution, are immune to this effect.

In addition, we feel that the following results are significant. First, we have shown that the packet method of data transmission is superior to the grouped technique of

passing data. On the same topic of data communication, our research indicates that the software overhead involved may well cancel out any benefit achieved from reducing either the amount of data sent or the number of transmissions needed to send it in multi-transputer systems programmed with Parallel C. We also learned that topology selection does not greatly influence the efficiency of multi-transputer implementations on the B008 of pure data parallel algorithms such as those we created for convolution. Also, our idea of making each transputer responsible for processing several rows of the image seems to work well, especially for implementing global techniques like moments and the Fourier transform. Lastly, we have created two new parallel algorithms for moment extraction which compare favourably with those previously suggested. We feel certain that our work has laid the foundations for an efficient, flexible multi-transputer image processing system including many techniques.

7.2 Future Work

It might be possible to improve the current implementations. For instance, one might try changing the priorities of the processes local to the source files as described in Section 2.6.1 in the multi-transputer programs. In addition, the memory ordering of the linked units might be changed to get the implementations to run faster. The direct link access model might be implemented. Alternatively, different data passing techniques might be attempted for the initial data pass.

Aside from these potential improvements, our system needs to be expanded. Other image processing techniques like the Fast Fourier Transform might be implemented. In addition, image processing applications that use the features extractable from convolution and regular moments could be programmed. For example, a contour tracing package might be created that uses one of our convolution programs as a front end processor for edge detection. A program that uses the regular moments extracted by one of our moment implementations to determine a set of moment invariants could

be implemented.

Lastly, the idea mentioned in the introduction of using transputers to model other parallel image processing technologies might be investigated.

References

[1] Bamieh, R. and De Figueiredo, R., 'A General Moment Invariants/Attributed-Graph Method for the Three Dimensional Object Recognition from a Single Image', *IEEE J. Robotics Automation*, Vol. 2, pp. 31-41, 1986.

[2] Belkasim, S. O., Shridhar, M. and Ahmadi, M., 'Pattern Recognition with Moment Invariants: a Comparative Study with New Results', *Pattern Recognition*, Vol. 24, No. 12, pp. 1117-1138, 1991.

[3] Brigham, E. O., *The Fast Fourier Transform*, Prentice Hall, N.J., 1974.

[4] Budrikis, Z. and Hatamian, M., 'Moment Calculations by Digital Filters', *AT & T Bell Lab Tech. J.*, Vol 63, No. 12, pp. 217-229, 1984.

[5] Casasent, D. and Cheatham, R., 'Image Segmentation and Real Image Tests for an Optical Moment Based Feature Extractor', *Optics Communs*, Vol. 51, pp. 227-230, 1984.

[6] Cok, R.S., *Parallel Programs for the Transputer*, Prentice-Hall, U.S.A., 1991.

[7] Chen, K., 'Efficient Parallel Algorithms for the Computation of Two-Dimensional Image Moments', *Pattern Recognition*, Vol 23, No. 12, pp. 109-119, 1990.

[8] Dudani, S., Breeding, K. and McGhee, R., 'Aircraft Identification by Moment Invariants', *IEEE Trans. Comput.*, Vol. 26, pp. 39-45, 1977.

[9] Gonzalez, R. C. and Wintz, P., *Digital Image Processing, 2nd ed.*, Addison-Wesley, U.S.A., 1987.

[10] Hatamian, M., 'A Real Time Two-dimensional Moment Generating Algorithm and its Single Chip Implementation', *IEEE Trans. Acoust. Speed Signal Process*, ASSP-34, pp. 546-553, 1986.

[11] Hoare, C., 'Communicating Sequential Processes', *Communications of the ACM*, Vol. 21, pp. 666-677, 1978.

[12] Hu, M. K., 'Pattern Recognition by moments invariants', *Proc I.R.E.*, Vol. 49, p. 1428, 1961.

[13] Kille, K., Ahlers, R.-J. and Schneider, B., 'Experiences with Transputer Systems for High Speed Image Processing', *Proc. SPIE - Int. Soc. Opt. Eng. (USA)*, Vol. 1386, pp. 76-83, 1991.

[14] Li, B. C. and Shen, J., 'Fast Computation of Moment Invariants', *Pattern Recognition*, Vol. 24, No. 8, pp. 807-813, 1991.

[15] LeBlanc, M., 'Image Processing on the PIP 1024', U.N.B. Report, 1992.

[16] Leu, J.G., 'Computing a Shape's Moments From its Boundary', *Pattern Recognition*, Vol 24, No. 10, pp. 949-957, 1991.

[17] Pan, Y., 'A Note on Efficient Parallel Algorithms for the Computation of Two-Dimensional Image Moments', *Pattern Recognition*, Vol. 24, No. 9, p. 917, 1991.

[18] Papoulis, A., *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, New York, 1965.

[19] Peng, A. and Kawamura, K., 'Parallel Image Processing on a Transputer-Based System', *Los Alamitos, CA, U.S.A.: IEEE Comput. Soc. Press*, pp. 235-238, 1990.

[20] *PIP Video Digitizer Board*, Product Manual, Matrox Systems, 1987.

[21] Pountain, D., 'A Personal Transputer', *Byte*, Vol. 13, No. 6, pp. 303-308.

[22] Prabhala, A., 'Digital RGB Camera and Interface for Optimum System Design', *Electronic Imaging '88*, Vol. 2, pp. 1141-1142, 1988.

[23] Reeves, A. P., 'Parallel Algorithms for Real-Time Image Processing', *Multicomputers and Image Processing, Algorithm and Program*, Academic Press, New York, pp. 7-18, 1982.

[24] Teague, M., 'Image Analysis via the General Theory of Moments', *J. Opt. Soc. Am.*, Vol. 70, pp. 920-930, 1980.

[25] Weitzman, C., *Distributed Micro/Minicomputer Systems Structure, Implementation, and Application*, Prentice-Hall, U.S.A., 1980.

VITA

Candidate's Full Name: Christopher John Turner

Date and Place of Birth: Feb. 1st, 1967, Hamilton, Ontario.

Permanent Address: 98 St. Clair Avenue, Hamilton, Ontario, L8M 2N5.

Schools Attended: Westdale Secondary School, Hamilton, Ontario 1980-1985.

Universities Attended: University of Toronto, 1985-1990, B.Sc.
University of New Brunswick 1990-1993.

Publication: Turner, C., Bhavsar, V. C. and Pochec, P., 'Multi-transputer Implementations of Some Feature Extraction Algorithms', *Proc. of International Conference on Robotics, Automation and Computer Vision '92 (ICARCV '92)*, Singapore, pp. CV 17.6.1 - CV 17.6.4, 15-18 Sept. 1992.